Chapter 1

Construindo um jogo MMORPG com Python

Roberto Fagá Jr, Vivian Genaro Motti, Maria da Graça Pimentel

Citação: FAGÁ JR, R. ; MOTTI, V. G. ; PIMENTEL, M. G. C. . Construindo um jogo MMORPG com Python. Sociedade Brasileira de Computação SBC, 2009 (Mini-Curso: IX ESCOLA REGIONAL DE COMPUTAÇÃO BAHIA-ALAGOAS-SERGIPE).

Abstract

The electronic game industry have been increasing in the last years, specially concerning online games. In order to develop a game, different knowledges are required. This chapter main goal is to create a 2D client of a MMORPG game showing some of these knowledges as well as coding each part of the game client. Some game development and network techniques are shown in this chapter. Python language is introduced and PyGame game library is used to make the proposed MMORPG client game.

Resumo

O mercado de jogos eletrônicos tem sido expandido nos últimos anos, sendo que jogos online têm apresentado um aumento ainda maior. A criação de um jogo envolve diversas áreas de conhecimento. O objetivo deste capítulo consiste na construção de um cliente 2D para um jogo MMORPG, para tal foi utilizada uma abordagem prática que descreve a construção do projeto. O capítulo introduz o participante a técnicas de desenvolvimento de jogos e de desenvolvimento de aplicações que utilizam redes de computadores. Introduz também a linguagem Python, e utiliza a biblioteca PyGame para a construção do jogo.

1.1. Introdução

A criação de um jogo é interessante. E não apenas pela diversão que um jogo pode proporcionar ao jogador, mas por diversos motivos, como o esforço e desafio nas diversas áreas da computação que são exigidos, o valor cultural que um jogo pode

conter dos desenvolvedores ou no qual os desenvolvedores focam e a maneira de entretenimento e treinamento de habilidades do jogador.

Os jogos eletrônicos, ou *games*, têm evoluído em sua complexidade e esforço de produção, sendo que diversos jogos comerciais atualmente demandam esforço maior do que a produção de filmes de Hollywood. Esforço que pode ser maior para a modalidade de jogos MMOG – *Multiplayer Massive Online Games*, na qual muitos jogadores participam jogando em um mesmo ambiente, criando mundos virtuais para milhares de jogadores.

Este capítulo trata da produção de um jogo MMORPG - *Multiplayer Massive Online Role Playing Game*, gênero de jogos que criam mundos virtuais para que os jogadores vivenciem personagens nesses mundos. O jogo é composto de cliente e servidor, sendo tratada no capítulo apenas a construção do cliente, pois é a que mais utiliza recursos de desenvolvimento de jogos.

A linguagem Python possui uma sintaxe simplificada que facilita sua aprendizagem, sendo uma linguagem completa e de escrita rápida. A Pygame é uma biblioteca em Python que contém módulos para auxiliar na construção de jogos.

As tecnologias utilizadas neste capítulo são Software Livre, um aspecto interessante para a produção de jogos e desenvolvimento de aplicações. O uso de Software Livre em jogos sofre algumas resistências, pois em geral o lucro financeiro das empresas de jogos são exclusivamente dos usuários com a venda de Software, enquanto que empresas de aplicações em Software Livre têm seu lucro distribuído entre usuários e outras empresas, além dele nem sempre estar vinculado à venda do produto, mas sim ao suporte ou à manutenção.

Analisado sob a perspectiva de compartilhamento de conhecimento, o uso de Software Livre para jogos é positivo, pois a demanda de tecnologias e algoritmos necessários para a produção de um jogo é muito elevada. As tecnologias livres têm grande importância para reduzir o custo da produção de um jogo, para melhorar a qualidade com tecnologias modernas e para compartilhar o conhecimento e as técnicas de criação de jogos.

Outra motivação para uso de Software Livre é que a produção de jogos comerciais do gênero MMORPG são de alto custo, passando dos 10 milhões de dólares (Carpenter, 2003). Investimentos de alto custo para produção de jogos, considerando o cenário brasileiro, não são comuns, sendo então um dos caminhos para a produção de jogos com qualidade e com custo reduzido o uso de tecnologias livres.

A criação de um cliente MMORPG será exposta neste capítulo, precedendo a Seção 1.2 com uma introdução à linguagem Python, com os tópicos necessários para entendimento do capítulo. A Seção 1.3 descreve mundos virtuais e o conceito de MMORPG. A partir da Seção 1.4 inicia-se a explicação sobre como produzir jogos, sendo primeiro introduzida a Pygame, seguida por bibliotecas de jogos em Python e então é apresentada a estrutura básica do jogo a ser criado. A Seção 1.5 cria interação do usuário, ou jogador, com o jogo. A Seção 1.6 traz uma introdução ao conceito de Redes computacionais, com a implementação da conectividade do cliente ao servidor, encerrando o jogo. Todos os passos do cliente bem como o servidor do jogo estão disponíveis no *link* <http://code.google.com/p/pygame-mmorpg/>.

1.2. Conhecendo Python

Muitas linguagens de programação demandam muito tempo de aprendizado, devido à alta complexidade que as mesmas possuem. Normalmente o aprendizado de algoritmos escritos em linguagem natural antecede o de linguagens de programação. A sintaxe da linguagem Python é muito semelhante a uma linguagem natural em formato de algoritmo, já que dispensa a utilização de chaves para demarcação de blocos, o uso de ponto-e-vírgula para fim de linha, entre outros fatores que serão expostos nesta seção.

Python é uma plataforma de programação voltada para a agilidade de produção de aplicações, contando com uma sintaxe simplificada e orientada a objetos (Python Foundation, 2009). Muitas vezes é classificada como uma linguagem de *script*[1] por ser ágil, entretanto possui todos os recursos para a construção de grandes aplicações (Reis, 2004). A utilização de uma linguagem como Python para a construção de jogos e de pequenas aplicações é interessante tanto para contextos educacionais como profissionais, e durante esse capítulo será possível observar a facilidade e agilidade para certas rotinas. Ainda, a plataforma bem como a maior parte das bibliotecas é Software Livre, o que garante a economia e a qualidade do código.

Devido a sintaxe próxima a de algoritmos, alguns programadores habituados com as linguagens C e Java podem achar o código fonte em Python confuso a primeira vista, já que o mesmo possui uma estrutura obrigatoriamente padrão, como a identação, enquanto que possui liberdades como a ausência de declaração de variável.

Lutz & Acher (2003, p.3) citam algumas vantagens da linguagem Python que motivam os desevolvedores a utilizarem-na, dentre elas destacam-se:

- ***Qualidade de Software***: O código Python foi desenvolvido para ser legível por humanos, além de ser simplificado. Por isso, a linguagem se torna fácil para manutenção e alterações.
- ***Produtividade***: O código fica em tamanho reduzido se comparado a código Java ou C, por consequência é necessário menos esforço para escrever, corrigir erros ou mantê-lo.
- ***Portabilidade***: Assim como outras linguagens multiplataforma, um código Python funciona em diferentes plataformas, sendo possível rodar um mesmo código em Windows, Linux ou outros.
- ***Bibliotecas***: Uma das vantagens em relação a outras linguagens ágeis (ou de *script*) é a grande coleção de bibliotecas que a plataforma Python tem. É possível encontrar bibliotecas que fazem quase todas as tarefas que são necessárias para uma aplicação, como é o caso de bibliotecas para jogos por exemplo. Muitas bibliotecas são apenas *bindings*, ou seja, apenas mapeiam funções para bibliotecas em código nativo do sistema operacional, garantindo ótimo desempenho.
- ***Componentes de integração***: É possível integrar código Python com outras linguagens e tecnologias, como C, C++, Java, .NET, Corba ou também utilizar interfaces como SOAP. Muitas aplicações utilizam abordagens híbridas, como em jogos, em que muitas vezes o desenvolvimento da parte crítica do jogo, que consome de fato muito processamento, é realizada com linguagens compiladas

1 Linguagens de script são normalmente assim chamadas por serem utilizadas para conectar aplicações ou automatizar tarefas.

nativamente como C e C++, e todo o restante que não utiliza grande processamento é feito com linguagens ágeis, como Python.

A identação, ou recuo que é realizado no código fonte para demarcar um bloco, é necessária em Python pois apenas ela demarca quando inicia e termina um bloco de alguma estrutura, como um *if* ou um *for*, bem como funções.

Não há necessidade também de declaração de variáveis, ou seja, não é necessário determinar previamente o tipo das mesmas. O tipo da variável é definido no momento de sua atribuição, de acordo com o tipo do valor a ser recebido (se é um texto, a variável é então do tipo texto, se número inteiro, é do tipo inteiro, e assim por diante). Diferentemente de outras linguagens, essas variáveis tratam-se apenas de ponteiros para objetos, o que permite que a uma mesma variável possa ser atribuído um texto em um momento e em outro ser atribuído um número inteiro.

O objetivo desta seção é preparar o leitor com noções básicas de Python. Para um aprendizado completo, é recomendada a leitura de um livro específico sobre a plataforma Python. Nas subseções a seguir serão introduzidos conceitos básicos da linguagem, operadores, estruturas de dados, funções e estruturas de controle, encerrando com alguns conceitos mais avançados.

1.2.1. Terminal interativo

Uma excelente ferramenta para aprendizado da linguagem é o terminal interativo, o qual deve ser a primeira coisa a ser executada depois de instalado o compilador Python. As instruções de instalação podem ser encontradas em diferentes páginas da Web, dependendo do sistema operacional que esteja sendo utilizado.

O terminal interativo (*interpreter interface* no Linux, *Python command line* no Windows) possibilita que a cada linha de código Python digitada, o sistema já o compile e execute na máquina, possibilitando correção de erros e aprendizado enquanto se desenvolve um código. A Figura 1 apresenta o terminal Python sendo executado em um terminal Linux, com o símbolo do terminal “>>>” (*prompt*) representando que o terminal está esperando por algum código.

```
faga@faga-desktop:~$ python
Python 2.5.2 (r252:60911, Oct  5 2008, 19:24:49)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figura 1: Iniciando o terminal interativo Python

Os elementos da estrutura da linguagem serão apresentados a seguir. É possível ainda escrever código em arquivos fonte (arquivos de extensão “.py”), e compilá-los depois com Python, e isso é necessário para fazer aplicações que demandam esforço maior do que os exemplos simples desta seção.

1.2.2. Operadores diversos

Como padrão de toda a linguagem quando está se aprendendo, abaixo segue o “olá mundo” (*hello world*), utilizando a função ***print***, que imprime tudo o que a segue, e uma cadeia de caracteres (*string*) sempre representada por aspas duplas ou aspas simples e utilizando também caracteres especiais comuns a outras linguagens, no caso o **\n** para quebra de linha e **\t** para tabulação.

```
>>> print "Olá mundo"
Olá mundo
>>> print '\tOlá mundo\n'
     Olá mundo
```

Assim que uma linha de código é inserida e encerrada por um símbolo indicador de fim de linha, o terminal responde com a impressão de algo e disponibiliza para entrada novamente assim que a execução da linha for encerrada. É possível também utilizar como calculadora, com soma, multiplicação, e divisão, sendo que / (barra) é utilizado para divisão comum, como 7/2 resulta em 3 por ser divisão de inteiros, e 7/2.0 resulta em 3.5 por ter um número de ponto flutuante em alguma parte da operação, além dos símbolos tradicionais de multiplicação (*****), soma (+), subtração (-) e módulo (**%**):

```
>>> 7+2
9
>>> 8*4-2
30
>>> 7/2
3
>>> 7/2.0
3.5
>>> 7%2
1
```

Existem em particular dois símbolos diferentes em relação a outras linguagens de programação: // (duas barras) para divisão com o valor chão (maior inteiro menor que o valor resultante) e ****** (dois asteriscos), para potência.

```
>>> 7//2.0
3.0
>>> 2**4
16
>>> 2**(4+1)
32
```

É interessante que muitos elementos são reaproveitados em diferentes objetos. Por exemplo, o elemento de soma (+) pode também concatenar *strings*, bem como (*****) multiplica **n** concatenações. A atribuição de variáveis é dinâmica, então basta criar um nome qualquer sem estar entre aspas e receber algum valor:

```
>>> variavel = "abc"*2
>>> variavel
'abcabc'
>>> "abc"+"def"
'abcdef'
```

Note que na atribuição não houve impressão na tela do valor resultante, pois o valor foi para a variável, e não para o terminal como na segunda linha. Para melhor entendimento, pode-se imaginar por hora que sempre há uma função ***print*** antes do código digitado.

A comparação de variáveis é realizada através do operador == ou da expressão ***is***, e dos símbolos > (maior)**,** < (menor), >= (maior ou igual), <= (menor ou igual). A

diferença entre os dois primeiros é que o operador == retorna a comparação se os valores são iguais, enquanto que a expressão ***is*** retorna se as variáveis apontam para o mesmo objeto, e essa diferença só será observada quando tratar objetos:

```
>>> a = 4
>>> b = 7*4
>>> a == b
False
>>> a > b
False
>>> 0 <= 0
True
>>> a == b/7
True
>>> "abc" == "abC"
False
>>> "abc" == "abc"
True
```

Os tipos ***True*** e ***False*** (note a primeira letra maiúscula) denominam o tipo *booleano* verdadeiro ou falso, respectivamente. Eles também são respostas para o operador ***in***, que retorna se um elemento está contido em outro, podendo ser utilizado em cadeias de caracteres:

```
>>> "abc" in "qabc"
True
```

Outro operador muito utilizado em códigos Python e presente em arquivos fonte é o #, utilizado para comentários. Tudo o que se segue na mesma linha do caractere # é ignorado pelo compilador Python.

1.2.3. Funções e módulos

A função ***print*** é especial do Python por não precisar de parênteses para ser utilizada, entretanto todas as outras necessitam, sendo três outras funções muito úteis quando se está aprendendo a linguagem: ***type***, ***dir*** e ***help***.

A função ***type*** retorna o tipo de um determinado objeto (variável), sendo útil já que em Python não se declara variável:

```
>>> type(1)
<type 'int'>
>>> type("abc")
<type 'str'>
```

Já as funções ***dir*** e ***help*** retornam, respectivamente, os métodos de um determinado objeto (as funções que um objeto pode ter) e os métodos, atributos e todos os detalhes sobre um objeto:

```
>>> dir(1)
['__abs__', '__add__', '__and__', '__class__', '__cmp__',
'__coerce__', '__delattr__', '__div__', '__divmod__', '__doc__',
'__float__', '__floordiv__', '__getattribute__', '__getnewargs__',
'__hash__', '__hex__', '__index__', '__init__', '__int__',
'__invert__', '__long__', '__lshift__', '__mod__', '__mul__',
```

```
'__neg__', '__new__', '__nonzero__', '__oct__', '__or__',
'__pos__', '__pow__', '__radd__', '__rand__', '__rdiv__',
'__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__',
'__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__',
'__rpow__', '__rrshift__', '__rshift__', '__rsub__',
'__rtruediv__', '__rxor__', '__setattr__', '__str__', '__sub__',
'__truediv__', '__xor__']
```

Os métodos que estão entre quatro *underscores* (__ __) são privados, não devem ser utilizados por chamadas, mas é possível de qualquer modo. Para chamar funções e métodos com mais de um parâmetro, basta separá-los por vírgula. Para invocar um método, utiliza-se um ponto entre o objeto que o possui e o método com os parâmetros, se houverem, como no uso do método ***find*** de uma *string*:

```
>>> "abc".find("c")
2
```

Outra característica fundamental de Python é o uso de módulos, que é a maneira de se importar bibliotecas. Com o uso da chamada ***import*** é possível importar alguns módulos, como o módulo ***os*** para questões do sistema operacional, dentre outros. Exemplo de uso:

```
>>> import os
>>> os.listdir('.')
['tango-icon-theme-extras', 'tango-icon-theme']
```

Durante este capítulo serão utilizados diversos módulos. Quando encontra-se um módulo desconhecido, usa-se o ***help(modulo)*** para exibir os módulos e variáveis no terminal interativo.

1.2.4. Estruturas de dados

Além dos tipos básicos (inteiro, ponto flutuante, *booleano* e *string*), Python também conta com alguns tipos de dados especiais, que são muito úteis durante a programação de aplicações, inclusive de jogos.

Em outras linguagens de programação, costuma-se utilizar vetores para fazer listas de elementos. Em Python, existem as listas (*list*), que funcionam de modo semelhante a vetores, mas com algumas particularidades. As listas são representadas por um conjunto de valores separados por vírgula, entre colchetes, e usa-se os colchetes também para acessar uma posição do vetor:

```
>>> lista = [1, "dois", 3.0, [4, 5]]
>>> lista[2]
3.0
```

As listas podem conter qualquer tipo de objeto, mesmo misturados. Elas também são dinâmicas, e permitem que operações de adição de elementos (***append***), remoção (***remove***), multiplicação (*****) e concatenação (**+**) possam ser realizadas:

```
>>> lista1 = [1,2]
>>> lista2 = [3,4,5]
>>> lista1 + lista2
```

```
[1, 2, 3, 4, 5]
>>> lista1*2
[1, 2, 1, 2]
```

A remoção e a adição de um elemento podem ser realizadas pelos métodos presentes nos objetos do tipo lista:

```
>>> lista1.append(9)
>>> lista1
[1, 2, 9]
>>> lista1.remove(9)
>>> lista1
[1, 2]
```

Além das listas, existem também as tuplas (*tuples*), as quais funcionam também como vetores, mas estáticos, ou seja, não podem sofrer alterações, mas podem ser utilizadas para resultarem em uma nova tupla:

```
>>> tupla = (1,2,3)
>>> tupla
(1, 2, 3)
>>> tupla + (1,2)
(1, 2, 3, 1, 2)
>>> tupla[2]
3
```

As tuplas podem ser utilizadas sem parênteses também, ou seja, quando houver valores separados por vírgula, o compilador Python já assume que trata-se de uma tupla:

```
>>> a, b = 1, 2
>>> a
1
>>> b
2
>>> print 1, 2
1 2
```

As *strings* também podem ser vistas como tuplas de caracteres, mas possuem algumas funções a mais. Isso significa que ambas, quando são concatenadas, por exemplo, são na realidade instanciadas em um novo objeto, enquanto que em listas, uma se altera localmente, sem instanciar um novo objeto por exemplo nesse caso:

```
>>> lista
[1, 'dois', 3.0, [4, 5], 9]
>>> lista += [1,2,3]
>>> lista
[1, 'dois', 3.0, [4, 5], 9, 1, 2, 3]
```

A maneira de acesso a listas, tuplas e cadeias de caracteres é com o uso de colchetes. A maneira apresentada até o momento consiste em passar o número referente ao índice, mas é possível também criar uma sublista, subtupla ou *substring* com os comandos:

```
>>> len(lista)
```

```
5
>>> lista[1]
'dois'
>>> lista[0]
1
>>> lista[0:1]
[1]
>>> lista[0:2]
[1, 'dois']
```

A função ***len*** utilizada na primeira linha retorna o tamanho da lista, e pode ser utilizada com qualquer objeto que seja de alguma variação de lista, como tuplas e dicionários. A maneira de acesso aos elementos com apenas um inteiro sempre retorna o elemento da posição, mas com o uso de **:** (dois pontos) é possível criar sublistas, sendo o primeiro elemento referente à posição inicial e o segundo à posição final (note que o elemento da posição final não é colocado na sublista). É possível ainda utilizar índices negativos para percorrer a lista do final para o começo ou utilizar espaço vazio para demarcar começo e fim:

```
>>> lista[0:-2]
[1, 'dois', 3.0]
>>> lista[-5:-2]
[1, 'dois', 3.0]
>>> lista[:]
[1, 'dois', 3.0, [4, 5], 9]
>>> lista[:-1]
[1, 'dois', 3.0, [4, 5]]
```

Outra estrutura muito utilizada é o dicionário, que funciona como uma lista, mas ao invés de índices numéricos, o acesso é por qualquer tipo de índice, por meio de uma chave, como *strings* e outros objetos. Para criar um dicionário, basta enumerar entre vírgulas pares chave:valor (entre dois pontos) entre chaves (**{ }**):

```
>>> dicionario = {1:2, "dois":4, 5.0:"cinco ponto zero"}
>>> dicionario[1]
2
>>> dicionario["dois"]
4
```

1.2.5. Estruturas de controle

Como em outras linguagens, Python possui estruturas de controle de código, como ***if***, ***while*** e ***for***. A identação em Python deve ser realizada obrigatoriamente para demarcar o bloco, como por exemplo identar dentro de um ***if*** para executar o código aninhado somente se a condição for verdadeira (entende-se como verdadeiro o tipo ***True*** ou algum valor que não seja zero ou vazio):

```
>>> if 1:
...     print "ok"
...
ok
>>> if 0:
...     print "ok"
...
```

```
>>> if []:
...     print "ok"
...
>>> if [1,2,3]:
...     print "ok"
...
ok
>>> if True:
...     print "ok"
...
ok
```

Para utilizar mais de uma condição, utilizam-se os operadores ***not***, ***and*** e ***or***:

```
>>> if 1 and 2 and not False:
...     print "ok"
...
ok
```

A estrutura ***while*** funciona de forma semelhante, mas executando o código aninhado enquanto a condição dada for verdadeira:

```
>>> while var > 0:
...     var-=1
...     print var
...
4
3
2
1
0
```

A estrutura ***for*** sempre se baseia em uma variável dentro de uma lista, em que o código aninhado executa uma variável assumindo cada objeto da lista. Por esse motivo, o mais básico ***for*** necessita da função ***range*** que retorna uma lista de números consecutivos, mas a maior vantagem é utilizar elementos em listas, nas quais a variável que segue a palavra *for* é a variável que recebe o valor da lista a cada passo:

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> for i in range(5):
...     print i
...
0
1
2
3
4
>>> lista = ["abc", "def", "qualquer", "coisa"]
>>> for elemento in lista:
...     print elemento
...
abc
def
qualquer
coisa
```

1.2.6. Tratamento de erros

O tratamento de erros em Python é um recurso poderoso, que possibilita tanto encontrar falhas no código como tratar possíveis erros diretamente no código. O terminal interativo exibe as possíveis mensagens de erro quando alguma linha de código é executada, como utilização de uma variável sem valor atribuído:

```
>>> variavel
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'variavel' is not defined
```

O *traceback* (listagem das últimas chamadas referentes ao erro) exibe na linha abaixo o local onde o erro ocorreu e na última linha o tipo de erro com uma descrição, em inglês, que quase sempre é bem objetiva. Os erros podem ser tratados automaticamente com a estrutura ***try*** / ***except*** na qual o compilador Python tenta executar a estrutura do ***try*** e caso não consiga, vai para a estrutura ***except***:

```
>>> lista = range(5)
>>> lista[4]
4
>>> lista[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> try:
...     lista[5]
... except IndexError:
...     print "deu problema"
...
deu problema
>>> try:
...     lista[5]
... except:
...     print "deu problema"
...
deu problema
```

É possível fazer diversos ***except*** com a classe de erro em seguida, para tratar diferentes erros que uma chamada pode realizar. Caso não se especifique uma classe, o ***except*** será executado em qualquer erro, caso não tenha sido tratado por um ***except*** anterior.

1.2.7. Considerações finais de Python

Até o momento o essencial de Python já está apresentado, mas para agilidade e encorajamento no desenvolvimento de aplicações Python, são ainda necessárias algumas dicas.

A identação demarca as estruturas de dados em Python, mas ela exige alguns cuidados. Por exemplo, o compilador só reconhece a identação como igual caso haja a mesma quantidade de espaços e tabulações (originadas por exemplo pela tecla *tab* do teclado). Então é recomendado seguir sempre um mesmo padrão, como dois espaços ou quatro espaços, ou sempre usar tabulação.

Existem ainda funções úteis como a função de leitura de arquivos ***file***. A função ***file*** abre um arquivo de dados no sistema e, de acordo com o segundo parâmetro, de escrita ou leitura, retornando um objeto que manipula o arquivo:

```
>>> handler = file("arquivo.dat")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'arquivo.dat'
>>> handler = file("arquivo.dat", "w")
>>> dir(handler)
['__class__', '__delattr__', '__doc__', '__enter__', '__exit__',
'__getattribute__', '__hash__', '__init__', '__iter__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__str__', 'close', 'closed', 'encoding', 'fileno', 'flush',
'isatty', 'mode', 'name', 'newlines', 'next', 'read', 'readinto',
'readline', 'readlines', 'seek', 'softspace', 'tell', 'truncate',
'write', 'writelines', 'xreadlines']
>>> handler = file("arquivo.dat", "w")
>>> handler.write("agora da certo")
>>> handler.close()
>>> handler = file("arquivo.dat")
>>> handler.readline()
'agora da certo'
>>> handler.close()
```

O parâmetro '**w**' define modo de escrita, mas poderia ser utilizado '**a**' para escrever no final de um arquivo já existente ou criar um novo caso não exista. Quando o segundo parâmetro é omitido, o arquivo é aberto como leitura, mas poderia ser utilizado também '**r**'. Em Python, quando omitimos algum parâmetro de uma função ou de um método, é assumido o valor padrão do mesmo. Para mais detalhes dos parâmetros, a função ***help*** pode ser utilizada.

Dentre os diversos métodos que o objeto retornado pela função ***file*** têm, os métodos ***read*** e ***readlines*** são utilizados para leitura do conteúdo do arquivo e ***write*** para escrita.

Para a criação de uma função, utiliza-se a palavra ***def*** para demarcar que é um bloco de uma função, seguido do nome da função a ser dado e, entre parênteses, uma lista dos parâmetros a serem dados, que são nomes para variáveis:

```
>>> def funcao(var1, var2):
...     print var1, var2
...
>>> funcao(1,2)
1 2
>>> funcao("oi", "tchau")
oi tchau
```

Além do terminal interativo é possível também colocar o código fonte em um arquivo de texto qualquer, e executá-lo com o compilador Python. Essa será a forma da construção do jogo, pois o código é razoavelmente grande para ser sempre escrito no terminal interativo a cada execução do jogo.

Para aprofundamento de conhecimentos em Python, é recomendada a leitura do livro de Lutz & Aschler (2003) ou também o manual disponível na Internet de Christian

Reis (2004). Esta seção teve por objetivo apenas introduzir os conceitos básicos da linguagem para a produção do jogo.

1.3. Introdução ao gênero MMORPG

MMO (*Massively Multiplayer Online)* se refere a jogos nos quais o desenvolvedor cria um mundo aberto onde os jogadores podem criar seus próprios pesonagens para conviverem com outros semelhantes. Esses jogos estimulam a criatividade e a comunicação entre os participantes, que podem criar novas identidades e comunidades virtuais para se relacionarem.

MMORPGs (*Massively multiplayer online role-playing games*) são emergentes na indústria de jogos computacionais e são um gênero bastante popular. Esses jogos existem desde a década de 90, mas ganharam popularidade especialmente nos últimos anos. Esse gênero de jogos atraiu um público grande, unindo jogadores de RPG à jogadores de ambientes virtuais.

MMORPG (*Massively Multiplayer Online Role Playing Game*) são jogos interativos online com espaços ilimitados nos quais os personagens interagem e convivem como se houvessem mundos paralelos. Um jogador assume uma identidade fictícia e controla as atividades de seu personagem em um cenário fictício, com objetivos claros, e com sistemas de níveis para compensar os jogadores (Kamienski et al., 2008).

Os jogos online permitem interações sociais entre os participantes por serem dinâmicos e permitirem que os personagens formem grupos, enfrentem inimigos e avancem no jogo em conjunto. A arquitetura de construção do jogo caracteriza ele, definindo quantos jogadores podem usá-lo simultaneamente, qual o controle do jogo que os desenvolvedores possuem e qual o esforço de implementação necessário antes de publicar o jogo.

Os principais problemas encontrados em jogos do tipo MMORPG estão relacionados à segurança, como trapaças no jogo para ganhar vantagens injustas, acesso e uso ilegal de outras contas de usuários (Achterbosch et al., 2008).

MMORPG se desenvolveram a partir de dois gêneros de jogos: MUDs (*multi-user dungeons)* e CRPGs *(computer role playing games)* que são jogos para um ou vários jogadores (Achterbosch et al., 2008). São exemplos de jogos do gênero MMORPG: World of Warcraft (Blizzard, 2008), Star War Galaxies (Lucasfilm Ltda., 2002), EverQuest (Sony, 2008), Ultima Online (Eletronic Arts Inc., 2008) e Final Fantasy XI (Square-Enix, 2008).

Há dois modelos de desenvolvimento de personagens que podem ser usados: sistemas baseados em classes ou em habilidades, no caso das classes os jogadores escolhe uma classe, ou grupo, de personagens ao criar o seu próprio, a classe define os pontos fortes e fracos do personagem; já no sistema baseado em habilidades durante a criação do personagem o jogador deverá definir seus atributos, as habilidades dos personagens podem evoluir, conforme eles avançam no jogo ou conforme eles as utilizam.

Jogos de RPG em computadores são jogados em uma rede local ou de um servidor centralizado, no entanto para serem considerados MMORPG é necessário que muitos jogadores estejam envolvidos.

1.4. Começando o jogo

A construção de um jogo é normalmente um processo longo e demorado, além de trabalhoso. Entretanto a idéia deste capítulo é apresentar um curso rápido, logo a construção do jogo é rápida.

Para a construção do jogo poderiam ser utilizadas diversas bibliotecas para jogos em Python, muitas excelentes, como algumas que merecem destaque: Pyglet (Alex Holkner, 2008), Panda (The Panda3D Development Team, 2008), Soya (SOYA3D, 2008), PyOgre (Python-Ogre, 2009) e PyODE (PyODE, 2007), apresentadas na Tabela 1.

Biblioteca	Características	Dependências	*Binding*
Pygame	Utiliza SDL com algumas funções adicionais para gráficos 2D	SDL	SDL
Pyglet	Utiliza OpenGL com o propósito de uma biblioteca para gráficos 2D, que pode integrar com 3D	OpenGL, LibAV caso utilize vídeos	OpenGL
Panda3D	É uma *engine* 3D escrita em C++, usada comercialmente com suporte a *shader*	Panda3D em C++	Panda3D em C++
Soya3D	*Engine* 3D pura em Python, que utiliza diversas bibliotecas em código nativo	OpenGL, OpenAL, GLEW, SDL, Cal3D, PIL, ODE	-
PyOgre	Bem utilizada em Computação Gráfica, utiliza OpenGL e DirectX	OGRE, OpenGL e/ou DirectX	OGRE
PyODE	*Engine* para cálculos de física em geral	ODE	ODE

Tabela 1: Comparação de bibliotecas para jogos em Python

A biblioteca que será utilizada durante o desenvolvimento será a Pygame (Pygame, 2008e), pois apresenta uma abordagem simples para produção de jogos, utilizando muitos recursos da linguagem Python. Entretanto, o uso de outras bibliotecas é interessante tanto para aprendizado como para comparação de vantagens de cada uma, mas não é esse o foco do capítulo.

A biblioteca Pygame apresenta uma série de funções e módulos Python para auxiliar na construção de jogos. Ela está disponível para diversas plataformas no site oficial[2], sendo utilizada através do módulo ***pygame.***

1.4.1. Hello World em Pygame

O primeiro jogo a ser construído é extremamente sem graça: consiste em uma janela preta, que não faz absolutamente nada. Mas é a estrutura básica de um jogo, um local

2 http://www.pygame.org/

para o mesmo aparecer. É recomendado que, quando importar o módulo ***pygame*** também seja invocado a função ***init()***, que inicializa os módulos da Pygame:

```
>>> import pygame
>>> pygame.init()
(6, 0)
```

A função retorna uma tupla de dois valores, sendo o primeiro de módulos da Pygame inicializados com sucesso e o segundo dos que apresentaram alguma falha. Caso isso ocorra, verifique novamente a instalação da Pygame.

Para os que já possuem experiência em criar janelas de jogos, sabem a dificuldade que é realizar tal tarefa em diversas linguagens. Entretanto, em Pygame é extremamente simples, através do módulo ***pygame.display*** (Pygame, 2008a):

```
>>> dir(pygame.display)
['Info', '_pygame_C_API', '__pygameinit__', '__doc__', '__file__',
'__name__', 'flip', 'get_active', 'get_caption', 'get_driver',
'get_init', 'get_surface', 'get_wm_info', 'gl_get_attribute',
'gl_set_attribute', 'iconify', 'init', 'list_modes', 'mode_ok',
'quit', 'set_caption', 'set_gamma', 'set_gamma_ramp', 'set_icon',
'set_mode', 'set_palette', 'toggle_fullscreen', 'update']
```

O módulo possui diversas funções para trabalhar com a tela em que o jogo está sendo executado. Por hora, apenas a função ***set_mode*** será explicada e utilizada. Essa função cria uma superfície de desenho da Pygame em uma janela (caso não seja em tela cheia), de acordo com os parâmetros passados, sendo o primeiro uma tupla de dois valores com a resolução da tela (em *pixels*):

```
>>> screen = pygame.display.set_mode((640, 480))
>>> screen
<Surface(640x480x32 SW)>
```

A função retorna um objeto do tipo *Surface*, que é uma superfície de desenho da Pygame. Na documentação disponível na Internet[3] é possível ver as opções do segundo parâmetro, que pode ser por exemplo para que o jogo seja executado em tela cheia. Quando esse código for executado, uma janela preta aparecerá na tela, que nem poderá ser fechada. Isso ocorre pois não foi associado o evento “fechar janela” para de fato fechar a janela.

O próximo passo será construirmos um *loop*, ou seja, a estrutura que repetirá infinitamente até que o jogo seja encerrado. A grande maioria dos jogos utiliza-se de *loops*, nos quais cada estrutura representa uma parte do jogo, e assim é possível desenhar os objetos necessários na tela, movimentar personagens, tratar eventos dos usuários e do jogo, dentre outras rotinas que precisam ser executadas constantemente em um jogo.

É necessário controlarmos a quantidade de quadros por segundo do *loop*, pois como os computadores atuais possuem alto poder de processamento, a jogabilidade pode ser prejudicada caso a velocidade do jogo fique muito rápida. Para isso, a Pygame conta com o módulo ***time***, o qual possui o objeto *Clock* para controlar a quantidade de quadros por segundo, dentre outras utilidades:

3 http://www.pygame.org/docs/ref/display.html#pygame.display.set_mode

```
>>> clock = pygame.time.Clock()
>>> type(clock)
<type 'Clock'>
```

Com o uso do método ***tick***, o objeto controla o número máximo de quadros, sendo o único parâmetro a quantidade máxima de quadros por segundo. Dessa maneira, agora surge a primeira aplicação, na qual será utilizada uma variável *running* para condicionar o bloco de *loop*, que por enquanto não possuirá final já que a aplicação ainda não trata o evento de fechar a janela. O código resultante, que é recomendado agora que seja colocado em um arquivo fonte separado, segue abaixo:

```
import pygame
pygame.init()
running = True
screen = pygame.display.set_mode((640,480))
clock = pygame.time.Clock()
while running:
     clock.tick(7)
```

Uma observação é que, quando executado o código, deverá ser fechado como aplicação “travada”. Agora a estrutura principal do jogo está funcionando. Apesar de ser apenas uma tela preta, o *loop* já está em funcionamento e controlado.

1.4.2. Carregando e salvando informações em um jogo

Para o jogo que está sendo construído, é necessário armazenar e carregar seu mapa, bem como suas imagens estáticas. O mapa do jogo é baseado em uma grande matriz de números inteiros, na qual cada número representa o tipo de imagem que aquele quadrado é. Dessa maneira, é preciso armazenar essa matriz de números em algum arquivo.

Aplicações em geral utilizam diferentes técnicas para armazenar e carregar informações. Uma delas é criar um formato próprio baseado em texto e construir um interpretador para processar essas informações de alguma forma. Outra seria utilizar a função ***eval***, que interpreta texto em objeto Python, nesse caso o texto pode estar em um arquivo:

```
>>> eval("1")
1
>>> type(eval("1"))
<type 'int'>
>>> type("1")
<type 'str'>
>>> type(eval("[1,2,3]"))
<type 'list'>
```

Entretanto, para este exemplo será utilizado o módulo ***cPickle***, um *binding* da biblioteca *Pickle* construído em C, que possui em média desempenho superior ao módulo ***pickle*** que utiliza apenas código Python.

O módulo ***cPickle*** armazena e carrega objetos Python em arquivos. Além dos tipos básicos de Python, como listas e dicionários, esse módulo também armazena objetos de classes novas, tornando-o útil para ser utilizado para salvar e carregar o estado atual dos jogos. Dois métodos serão úteis neste capítulo: ***dump*** e ***load*** (Doll,

2002). O método ***load*** carrega as informações de um arquivo dado como parâmetro, enquanto que o método ***dump*** armazena no segundo parâmetro (arquivo) o objeto passado como primero parâmetro:

```
>>> lista = [1.0, 2, "tres"]
>>> import cPickle
>>> cPickle.dump(lista, file("arquivo.dat", "w"))
>>> nova_lista = cPickle.load(file("arquivo.dat"))
>>> nova_lista
[1.0, 2, 'tres']
>>> nova_lista is lista
False
>>> nova_lista == lista
True
```

Existe ainda um terceiro parâmetro para o método ***dump***, chamado de protocolo, que varia de 0 a 2. O protocolo zero determina que o arquivo resultante esteja em texto pleno, tornando-o compatível com diversas plataformas, enquanto que quanto maior for o valor, mais eficiente será a leitura e escrita e menos compatível com outras versões do ***cPickle***.

A utilidade dessa biblioteca é para carregarmos o mapa do jogo, de maneira rápida e fácil. Por possibilitar carregar e salvar variáveis Python em arquivos, é possível por exemplo salvar preferências do usuário, salvar o estado do jogo, dentre diversas outras operações que exigem armazenamento de dados.

A única ressalva é que o formato é facilmente editado em um bloco de notas qualquer, e portanto deve-se ter atenção evitando possíveis alterações nos arquivos pelo usuário para trapacear no jogo. Enquanto que para alguns possibilitar trapaças pode tornar o jogo ao jogador mais interessante, para outros pode tornar o jogo muito fácil, por consequência entediante para o jogador.

1.4.3. Criando mapa baseado em quadros

Em jogos de RPG, um importante componente é o mapa do jogo. O mapa é o mundo em que o personagem viverá e, tirando alguns jogos de RPG, a maior parte deles necessita de mapas extensos, para o jogador explorar um universo diferente. Quando trata-se de MMORPGs talvez tal importância aumente, pois diversos jogadores (até milhares) habitam um mesmo mapa, um mesmo mundo, e é necessário espaço para todos, seja para habitar, perseguir inimigos ou encontrar tesouros.

Pouquíssimos jogos de RPG, em especial MMORPG, utilizam como mapa uma só imagem representando o mundo inteiro. Normalmente os mapas são representados por uma matriz de valores, na qual cada elemento determina como é uma parte do mapa.

No caso de mapas tridimensionais, uma abordagem possível é representar uma matriz de elevação do terreno, para as alturas de cada ponto, e traçar polígonos dessa maneira. Depois, colocar objetos tridimensionais, como casas, árvores, etc, ou seja, os detalhes e objetos diferentes de terreno. Então utilizam-se texturas para preencher os polígonos, a fim de oferecer um aspecto mais realista ao cenário.

Pode-se considerar também o uso de texturas em um mapa com altura uniforme, simplificando o modelo de representação. A Figura 2 mostra um pedaço de um mapa dividido em 15 quadros, chamados de *tiles*. Os quadros A e B são das mesmas imagens,

bem como C e D, mas por estarem posicionadas de maneira apropriada geram um mapa contínuo.

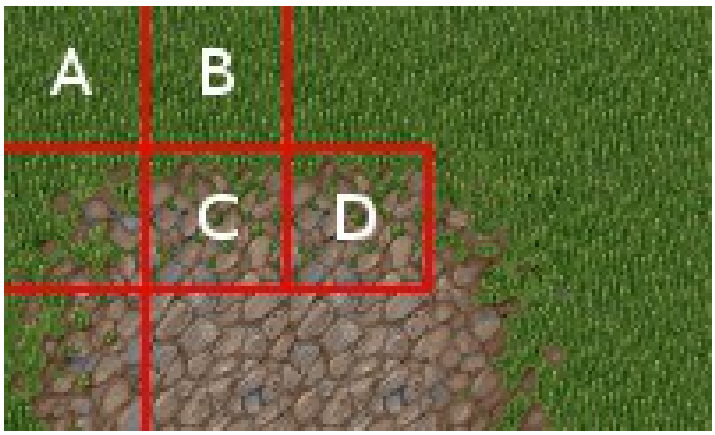

Figura 2: Imagens repetidas, chamadas de texturas, combinam-se com repetições para gerar o terreno de um mapa.

É possível também utilizar *tiles* para montar objetos no cenário, como casas por exemplo, ou qualquer outro objeto que possibilite repetição de imagens pequenas. Quanto menos uniforme for o objeto, pior será a representação em quadros, pois serão necessárias muitas imagens diferentes.

Após a criação do terreno, posicionam-se objetos pelo mapa, como árvores e pedras por exemplo. A Figura 3 mostra um mapa que utiliza um conjunto de texturas, já com árvores e pedras. Observando em detalhes, é possível ver a repetição dos quadros no lado direito da imagem.

Figura 3: Mapa construído com texturas repetidas, árvores e pedras.

1.4.4. Carregando o mapa do jogo

Para facilitar seu desenvolvimento, o jogo apresentado neste capítulo não utilizará altura em seu mapa, mas apenas qual imagem deve aparecer em cada quadro, gerando o mapa todo em uma única altura. Dessa maneira, utilizando apenas uma matriz de números inteiros é possível armazenar e construir o mapa em tempo real, durante o jogo, possibilitando economia de recursos computacionais, como espaço em disco utilizado e processamento (custa-se muito mais processar uma imagem muito grande do que diversas pequenas, já que dessa maneira é processada apenas a região da tela do jogador).

A dimensão da matriz (tamanho) é o total de quadros que o mapa tem, que pode ser multiplicado pelo tamanho de cada quadro para se ter idéia do total de pixels[4] do mapa. Para o jogo em construção, já existe um mapa construído, através de um programa que gerou-o aleatoriamente, e pode ser carregado no diretório “data”:

```
>>> import cPickle
>>> tilemap = cPickle.load(file("data/tilemap.txt"))
>>> len(tilemap)
500
>>> len(tilemap[1])
500
```

4 Pixels são pontos de uma imagem, na qual cada ponto assume uma cor a fim de gerar com vários pontos de várias cores.

O mapa presente no arquivo “tilemap.txt” possui uma matriz de inteiros variando de 0 a 23, na qual cada número é o correspondente a imagem “tileN.png” do diretório “imagens”. Dessa maneira, o arquivo tile0.png é o que representa o tile de código 0 da matriz, e assim por diante.

Apesar de facilitar o entendimento, seria mais eficiente se a matriz fosse representada como uma cadeia de caracteres por exemplo, na qual cada letra, número ou componente da cadeia representasse uma imagem, pois listas são objetos Python muito mais custosos para serem criados e manipulados do que cadeias de caracteres.

Para representar a posição de árvores, casas e pedras, é utilizado um outro arquivo de mapa, o “objmap.txt”. Também de dimensão 500 por 500 quadros, esse arquivo representa se há algum objeto em cada posição:

```
>>> objmap = cPickle.load(file("data/objmap.txt"))
>>> objmap[0][0]
255
>>> len(objmap)
500
```

O mapeamento dos números funciona de maneira análoga aos arquivos “tileN.png”, mas agora “objN.png”, sendo o número 255 representante de espaço vazio no mapa (apenas por convenção, idealmente poderia ser 0 ou -1).

Existe ainda o objeto 254 que representa que o espaço do mapa está ocupado mas não utiliza alguma imagem. Isso é feito para objetos que ocupam mais do que uma posição do mapa, como uma árvore que normalmente ocupa duas posições, sendo uma delas então representando a árvore e a outra um espaço ocupado. Poderia representar apenas a posição que o objeto começa e o próprio jogo trata as outras posições a serem ocupadas, mas dessa maneira facilita a movimentação do jogador e tratamento de colisões.

A quantidade de posições do mapa de objetos e de texturas é a mesma, a fim de facilitar o desenvolvimento, mas muitas vezes isso pode ser diferente, sendo interessante que haja mais posições para o jogador se situar e se locomover. Deve-se ressaltar que se tratando de um MMORPG, um mapa com muitas posições influencia diretamente no tráfego de dados entre clientes e servidor, já que quanto mais posições o mapa possui maior é a coordenada de cada objeto para que o servidor e o cliente se comuniquem.

O próximo passo agora é carregar as imagens para montar o mapa e os objetos, e então renderizar[5] as imagens na tela.

1.4.5. Carregando e exibindo imagens na tela

O módulo ***pygame.image*** (Pygame, 2008d) possibilita carregar imagens de formatos diversos (como PNG, JPEG e BMP) em uma *Surface*, uma superfície de desenho da Pygame. Dessa maneira, é possível carregar as texturas e os objetos a serem utilizados no Jogo. Não é o caso do jogo em construção neste Capítulo, mas é possível também utilizar imagens para interface gráfica, por exemplo.

Para carregar imagens no formato Pygame utiliza-se a função ***load*** do módulo ***pygame.image***:

---

5 Renderizar é o ato de transformar os dados em imagem final, imprimindo-o na tela por exemplo.

```
>>> import pygame
>>> pygame.image.load(file('imagens/tile0.png'))
<Surface(40x40x24 SW)>
```

É necessário apenas um parâmetro, o arquivo da imagem. Existem ainda funções para editar as imagens e salvá-las, que podem ser estudadas com a função ***help(pygame.image)***.

Para o jogo é necessário carregar todas as imagens a serem utilizadas. Por questões de desempenho utiliza-se muito carregamento dinâmico, no qual apenas as imagens a serem utilizadas em um momento do jogo são carregadas na memória. No jogo deste capítulo pode-se manter todas as imagens em memória pois são poucas, isso é algo a ser gerenciado quando há muitas imagens ou limitações no uso de recursos computacionais.

Ao invés de carregar imagem por imagem, esse procedimento pode ser feito com uma estrutura ***for***:

```
>>> tile = []
>>> obj = []
>>> for i in range(24):
...     tile +=[pygame.image.load( file('imagens/tile%s.png'%i) )]
...     obj +=[pygame.image.load( file('imagens/obj%s.png'%i) )]
...
>>> tile
[<Surface(40x40x24 SW)>, <Surface(40x40x32 SW)>, <Surface(40x40x32
SW)>, <Surface(40x40x32 SW)>, <Surface(40x40x32 SW)>,
<Surface(40x40x32 SW)>, <Surface(40x40x32 SW)>, <Surface(40x40x32
SW)>, <Surface(40x40x32 SW)>, <Surface(40x40x32 SW)>,
<Surface(40x40x32 SW)>, <Surface(40x40x32 SW)>, <Surface(40x40x32
SW)>, <Surface(40x40x32 SW)>, <Surface(40x40x32 SW)>,
<Surface(40x40x32 SW)>, <Surface(40x41x32 SW)>, <Surface(40x40x32
SW)>, <Surface(40x40x32 SW)>, <Surface(40x40x32 SW)>,
<Surface(40x40x32 SW)>, <Surface(40x40x32 SW)>, <Surface(40x40x32
SW)>, <Surface(40x40x32 SW)>]
```

O que acontece no código exposto é que uma lista é inicializada vazia (***tile = []*** e ***obj = []***) e o laço é executado com a variável ***i*** variando de 0 a 23. Cada execução concatena a lista anterior com uma nova lista contendo um elemento, em que cada interação do laço carrega o arquivo com nome “tileN.png” ou “objN.png”, onde N é o número que utiliza a tag “%s”. Tal tag faz com que o valor nessa posição seja da variável ***i***, presente logo após a *string* conectada com um sinal de módulo (%).

Uma vez carregada uma imagem em um objeto de superfície (*Surface*), este pode ser combinado com outras superfícies, como a tela principal do jogo ou outras imagens carregadas, através do método ***blit***. O método coloca a superfície passada no primeiro parâmetro sobre a própria superfície, na posição fornecida pelo segundo parâmetro em *pixels*:

```
>>> pygame.init()
(6, 0)
>>> screen = pygame.display.set_mode((640,480))
>>> screen.blit(tile[0], (0,0))
<rect(0, 0, 40, 40)>
```

É possível que nenhuma alteração seja exibida pelo terminal Python já que o processo do jogo não está em execução ainda. Portanto é preciso editar em um arquivo fonte para execução em modo cliente, também pelo tamanho e pela complexidade atuais do código. Pode-se também fazer um laço infinito no terminal para funcionar, pois quando o terminal é bloqueado para um laço de atualizar a tela, esta é atualizada corretamente.

Como o segundo parâmetro do método ***blit*** das superfícies é a posição, é necessário fazer no *loop* do jogo uma maneira de atualizar sempre toda a tela, cada objeto em sua posição correta. Faz-se uso então de três constantes:

```
>>> WIDTH, HEIGHT, TILE = 16, 11, 40
```

As constantes (***WIDTH***, ***HEIGHT***, ***TILE***) são referentes à largura e à altura em quadros, e ao tamanho em *pixels* de cada quadro *(tile*). O tamanho em *pixels* é o tamanho de cada *tile*, e como tratam-se de 640 *pixels* na horizontal, cabem então 16 quadros de 40 *pixels* (16 x 40 = 640 *pixels*). Na vertical cabem 12 quadros, mas por hora serão utilizados 11 quadros apenas, sendo o último uma área da tela reservada para impressão de textos de interação entre o jogador e o jogo.

Pode-se resumir que a tela é dividida em uma matriz de ***WIDTH*** por ***HEIGHT*** posições, no caso 16 x 11 posições, na qual cada posição é um quadro de imagem. O *loop* do jogo deve então percorrer cada uma dessas posições para desenhar o quadrado correspondente, primeiro sendo as texturas do mapa (*tilemap*) depois os objetos (*objmap*), pois os objetos devem ser renderizados depois para ficarem por cima do terreno:

```
>>> while 1:
...   for y in range(HEIGHT):
...     for x in range(WIDTH):
...       screen.blit(tile[1], (x*TILE, y*TILE))
...   pygame.display.flip()
...
```

No caso, em um *loop* de jogo, o que o trecho acima faz é imprimir o tile[1] (uma textura de gramado) em todas as posições da tela, e a função ***flip*** do módulo *display* atualiza toda a tela, ou seja, renderiza todos os objetos na tela a cada chamada. Para finalizar esse *loop* infinito a combinação de teclas “CTRL + C” finaliza o *loop*. Nota para a posição “(***x*TILE, y*TILE***)”, que retorna a posição (x,y) da matriz 16x11 posições, e multiplica o tamanho do quadro para obter-se o valor da posição em *pixels* na tela.

Idealmente os objetos poderiam ser renderizados também em *tiles*, ou seja, em quadros. Entretanto os objetos estão em imagens inteiras, o que é um pouco mais lógico do ponto de vista de armazenamento das imagens e de renderização na tela já que o objeto é manipulado por inteiro.

O método ***blit*** renderiza uma superfície na outra considerando o ponto de coordenada (0,0) a parte superior esquerda da superfície de parâmetro, ou seja, para imagens maiores do que um *tile* o restante da imagem se sobrepõe aos quadros abaixo e da direita ao quadro que se passou de parâmetro para renderização. A Figura 4 mostra o maior objeto presente no jogo, um pedaço de uma casa, que ocupa 4 por 6 quadros.

Figura 4: Pedaço de uma casa do jogo, o objeto possui o tamanho de 4 quadros horizontais e 6 verticais

Por esse motivo, é necessário que o *loop* do jogo renderize em posições negativas, a fim de garantir que a renderização cubra o extremo esquerdo da tela, como mostra o a Figura 5. Sendo o maior objeto 6 x 4 quadros, é necessário que o *loop* renderize uma matriz começando da posição negativa -6 x -4.

Figura 5: Renderização em posição negativa de objetos maiores que um quadro.

Dessa maneira, o *loop* do jogo deve começar em posições negativas apenas para os objetos, além de só renderizar objetos com ID menor que 24, já que só existem imagens de objetos até o número 23 (poderiam haver mais objetos no jogo). O *loop* então fica da seguinte maneira:

```
while running:
  clock.tick(7)
  for y in range(HEIGHT):
    for x in range(WIDTH):
      screen.blit(tile[tilemap[y][x]], (x*TILE, y*TILE))
```

```
  for y in range(-6, HEIGHT):
    for x in range(-4, WIDTH):
      if objmap[y][x] < 24:
        screen.blit(obj[objmap[y][x]], (x*TILE, y*TILE))
  pygame.display.flip()
```

Ocorrem duas estruturas ***for*** dentro do *loop*: a primeira para renderizar as texturas do mapa, a segunda, que começa em índices negativos para garantir a renderização do maior objeto, renderiza apenas objetos que sejam de índice menor que 24.

1.4.6. Carregando o mapa do jogo na tela

Tendo todo o necessário agora para a criação do cliente, e juntando todas as seções anteriores, o jogo construído que exibe o mapa carregando-o de arquivos é exibido a seguir:

```
import pygame, cPickle
pygame.init()

tilemap = cPickle.load(file('data/tilemap.txt', 'r'))
objmap = cPickle.load(file('data/objmap.txt', 'r'))
running = True
screen = pygame.display.set_mode((640,480))
clock = pygame.time.Clock()

WIDTH, HEIGHT, TILE = 16, 11, 40

tile, obj = [], []

for i in range(24):
  tile += [pygame.image.load(file('imagens/tile%s.png'%i))]
  obj += [pygame.image.load(file('imagens/obj%s.png'%i))]

while running:
  clock.tick(7)
  for y in range(HEIGHT):
    for x in range(WIDTH):
      screen.blit(tile[tilemap[y][x]], (x*TILE, y*TILE))
  for y in range(-6, HEIGHT):
    for x in range(-4, WIDTH):
      if objmap[y][x] < 24:
        screen.blit(obj[objmap[y][x]], (x*TILE, y*TILE))
  pygame.display.flip()
```

Uma imagem estática do mapa com algumas árvores será exibida em uma janela, que não irá se fechar sem ser manualmente.

1.5. Criando interação entre o jogo e o jogador

A interação dos jogos com os jogadores pode ser considerada como parte da área de Interação Usuário-Computador, responsável por estudar aspectos de *design*, avaliação e implementação de sistemas computacionais interativos para uso humano e fenômenos a eles relacionados. A usabilidade se insere nesse contexto, sendo uma característica que indica a facilidade, ou seja, quanto um usuário é capaz de interagir com um sistema ou um produto.

Problemas de usabilidade que dificultam a interação do usuário ou levam a erros, podendo mudanças no *design* levar a melhorias nesses aspectos. Aos problemas são atribuídas diferentes severidades, alguns podem representar barreiras potenciais para os usuários, pois não só dificultam a interação, mas impedem-na (Rocha & Baranauskas, 2003).

A influência da qualidade de interação pode ser percebida diretamente na jogabilidade, em que um jogo pode não ser agradável a um jogador se sua interação for ruim, como resposta atrasada de ações que o jogador realiza. Ainda, há o estudo de diferentes dispositivos de interação, permitindo maior realidade ao jogo ou maior jogabilidade.

1.5.1. Eventos na Pygame

Um dos pontos que a usabilidade em jogos trata é a captura das ações do jogador, que no jogo em construção é a captura de ações no mouse e no teclado. O módulo ***pygame.event*** (Pygame, 2008b) trata desses tipos de eventos, funcionando como uma fila de eventos, em que cada evento capturado é adicionado a uma lista Python. A função get retorna essa lista, deixando a fila da Pygame vazia novamente para novas ações:

```
>>> import pygame
>>> pygame.init()
(6, 0)
>>> pygame.event.get()
[]
```

No caso, pelo terminal interativo inicializado, virá uma lista vazia pois não há nem sequer uma janela criada. Então, inicializando a janela e pressionando algumas letras do teclado na janela, o comando retornará as teclas pressionadas bem como ações do mouse:

```
>>> screen = pygame.display.set_mode((640,480))
>>> pygame.event.get()
[<Event(3-KeyUp {'scancode': 0, 'key': 300, 'mod': 4096})>,
<Event(1-ActiveEvent {'state': 2, 'gain': 1})>, <Event(2-KeyDown
{'scancode': 38, 'key': 97, 'unicode': u'a', 'mod': 4096})>,
<Event(3-KeyUp {'scancode': 38, 'key': 97, 'mod': 4096})>, <Event(2-
KeyDown {'scancode': 56, 'key': 98, 'unicode': u'b', 'mod': 4096})>,
<Event(3-KeyUp {'scancode': 56, 'key': 98, 'mod': 4096})>, <Event(2-
KeyDown {'scancode': 54, 'key': 99, 'unicode': u'c', 'mod': 4096})>,
<Event(3-KeyUp {'scancode': 54, 'key': 99, 'mod': 4096})>, <Event(1-
ActiveEvent {'state': 2, 'gain': 0})>, <Event(3-KeyUp {'scancode':
0, 'key': 300, 'mod': 4096})>]
```

A lista contém objetos do tipo ***Event***, uma classe da Pygame que simboliza um evento. Cada objeto dessa classe tem atributos, os quais possuem um tipo e atributos variáveis de acordo com o tipo de evento. O atributo ***type*** determina o tipo, um valor numérico que é representado por uma constante conforme listagem na documentação oficial (Pygame, 2008b) :

```
>>> eventos = pygame.event.get()
>>> evento = eventos[0]
>>> evento
```

```
<Event(1-ActiveEvent {'state': 4, 'gain': 1})>
>>> evento.type
1
>>> evento.state
4
>>> evento.gain
  1
```

A impressão é feita em um formato de dicionário Python, mas cada atributo pode ser acessado diretamente pelo objeto. Ao invés de utilizar os códigos numéricos de cada tipo de evento, pode-se utilizar as constantes da Pygame, bem como em outras partes do código. Para isso, pode-se colocar no começo do jogo a linha:

```
>>> from pygame.locals import *
```

Dessa maneira, todas as constantes da Pygame são importadas. É interessante importar todas as constantes, já que para fins didáticos é mais simples importarmos tudo e não somente as que utilizaremos. Constantes em Python são, por convenção, representadas em caixa alta, mas são variáveis de qualquer forma.

A função ***get*** do módulo ***pygame.event*** pode ter como parâmetro também o tipo de evento, para filtrar por exemplo apenas quando um botão do teclado é pressionado, retirando apenas os eventos desse tipo da fila de eventos da Pygame:

```
>>> pygame.event.get(KEYDOWN)
[<Event(2-KeyDown {'scancode': 38, 'key': 97, 'unicode': u'a',
'mod': 4096})>, <Event(2-KeyDown {'scancode': 56, 'key': 98,
'unicode': u'b', 'mod': 4096})>, <Event(2-KeyDown {'scancode': 54,
'key': 99, 'unicode': u'c', 'mod': 4096})>]
```

Pode-se também criar um evento, através da classe ***Event***, e colocá-lo(o ? evento ou classe) na fila com a função ***post***:

```
>>> e = pygame.event.Event(KEYDOWN, {'key': 99})
>>> pygame.event.post(e)
>>> pygame.event.get(KEYDOWN)
[<Event(2-KeyDown {'key': 99})>]
```

O primeiro parâmetro de um evento é o tipo, que pode ser utilizado por uma das constantes ou um valor numérico, e o segundo parâmetro é um dicionário com os atributos e seus valores. Dessa maneira é possível criar eventos personalizados:

```
>>> e = pygame.event.Event(25, {'ano': 2009})
>>> pygame.event.post(e)
>>> pygame.event.get(25)
[<Event(25-UserEvent {'ano': 2009})>]
>>> e.ano
2009
```

1.5.2. Tratando eventos da Pygame

Durante o *loop* principal do jogo, é necessário tratar os eventos a cada quadro por segundo, para garantir que as ações sejam tomadas no mesmo instante em que a imagem é renderizada. Devido ao formato da Pygame ser em fila, não ocorrerão problemas de

travar o jogo em espera a algum comando, mas com outras bibliotecas é necessário cautela para que o *loop* não fique preso esperando alguma ação do jogador, fazendo com que o jogo "congele" a tela enquanto não realiza alguma ação.

Como não existem estruturas ***which/case*** em Python, será necessário fazer uma estrutura utilizando ***if/else*** para tratar os diferentes tipos de eventos. Adicionando então o tratamento de evento do tipo ***QUIT***, que representa quando tentou-se fechar a janela, o *loop* do jogo fica da seguinte maneira:

```
from pygame.locals import *
while running:
  clock.tick(7)
  for e in pygame.event.get():
    if e.type == QUIT:
      running = False
  for y in range(HEIGHT):
    for x in range(WIDTH):
      screen.blit(tile[tilemap[y][x]], (x*TILE, y*TILE))
  for y in range(-6,HEIGHT):
    for x in range(-4,WIDTH):
      if objmap[y][x] < 24:
        screen.blit(obj[objmap[y][x]], (x*TILE, y*TILE))
  pygame.display.flip()
```

Em que uma estrutura ***for*** cuida de cada evento retirado da fila, passando pelo teste do ***if/else*** em relação ao tipo do evento e tratando-o. Com o *loop* acima, finalmente o jogo fecha quando se deseja fechá-lo. É possível também utilizar diversas funções ***get*** com diferentes parâmetros, e isso torna possível tratar tipos de eventos em momentos diferentes, como por exemplo eventos de ação do jogador em 7 quadros por segundo mas tratar fechamento da janela instantaneamente (seria necessário um *loop* em paralelo ou contagem distinta de quadros por segundo).

Existem ainda alguns eventos personalizados que serão utilizados no jogo:

```
GOTO = 24
MESSAGE = 25
SETOBJECT = 26
SYSTEM = 27
GETINFO = 28
```

Cada tipo de evento será utilizado no jogoº O ***GOTO*** é um evento de deslocamento do personagem, ***MESSAGE*** é usado para mensagens ao jogador, ***SETOBJECT*** cria ou destrói um objeto em uma coordenada, ***SYSTEM*** é usado para mensagens do sistema (como erros) e usando ***GETINFO*** obtém-se informações do servidor de um objeto no mapa.

1.5.3. Movimentando a tela do jogo

Para movimentação da tela do jogo, é necessário que a matriz de renderização se desloque pelos mapas, tanto de *tiles* quanto de objetos. A matriz de renderização, que também pode ser chamada de matriz de deslocamento, representa os pontos do mapa e dos objetos que devem ser renderizados na tela, sendo ela apenas uma parte do mapa, pois é inviável em termos de processamento renderizar todo o mapa fora da tela (além de ser um desperdício de recursos).

Não é necessário criar a cada iteração do *loop* uma nova matriz de renderização baseada na posição do jogador. Pode-se apenas utilizar variáveis de deslocamento referentes às matrizes originais, do mapa e dos objetos, sendo que o deslocamento é a movimentação do jogador, pois o que deve ser renderizado é a região ao redor do jogador.

Como o jogador deve se situar no centro da tela, é interessante definir uma coordenada referente ao centro para facilitar o desenvolvimento, subtraindo 1 de cada valor pois o personagem ocupa mais do que um quadro, além das variáveis ***slide_x*** e ***slide_y*** para denominar o deslocamento da tela (horizontal e vertical respectivamente) com valor inicial de 0 (começando no canto superior esquerdo do mapa):

```
MIDDLE = WIDTH/2 - 1 , HEIGHT/2 – 1
slide_x = 0
slide_y = 0
```

O motivo da subtração na constante ***MIDDLE*** é puramente estético, quando o jogo estiver em fase final essa constante pode ser alterada para ver os diferentes posicionamentos do personagem do jogador.

Para movimentar o mapa, será criada a função ***goto***, a qual desloca o mapa para as coordenadas (x,y) passadas como parâmetro, referentes à posição do jogador. Por se tratarem de variáveis globais, que pertencem a todo o código, é necessário declarar que as variáveis já inicializadas ***slide_x*** e ***slide_y*** são globais. Então deve-se posicioná-las na coordenada de parâmetro (x,y) e subtrair o ponto central, pois o deslocamento é do personagem e a visualização deve ser realizada no centro da tela:

```
def goto(x, y):
  global slide_x, slide_y
  slide_x = x - MIDDLE[0]
  slide_y = y - MIDDLE[1]
  pygame.display.set_caption("MMORPG Client - Pos: %2d, %2d"%(x, y))
```

A função ***set_caption*** muda o título da tela, nesse caso imprimindo a posição X,Y do personagem. A Figura 6 mostra a matriz do mapa e a tela do jogo ocupando apenas uma parte, o que seria a matriz de renderização ou deslocamento. As variáveis ***slide_x*** e ***slide_y*** deslocam a matriz do ponto (0,0) do mapa (representado por B) até o canto superior esquerdo da tela.

A constante ***MIDDLE***, composta por dois valores, é referente ao deslocamento do personagem, ponto B, em relação ao canto da tela. Isso é necessário pois tem-se o objetivo de renderizar a região ao redor do personagem, colocando-o no centro da tela. Porém o jogo deve tratar a posição do jogador e não do canto da tela para tratamento de colisões com outros objetos (o que colide é o jogador, e não o canto da tela).

A posição do personagem no mapa será sempre dada pela coordenada (***slide_x*** + ***MIDDLE[0]*** , ***slide_y*** + ***MIDDLE[1]***). O valor do deslocamento da tela pode também ser negativo, nesse caso deslocando a matriz de renderização da direita para a esquerda e de cima para baixo, de acordo com o eixo negativo da Pygame.

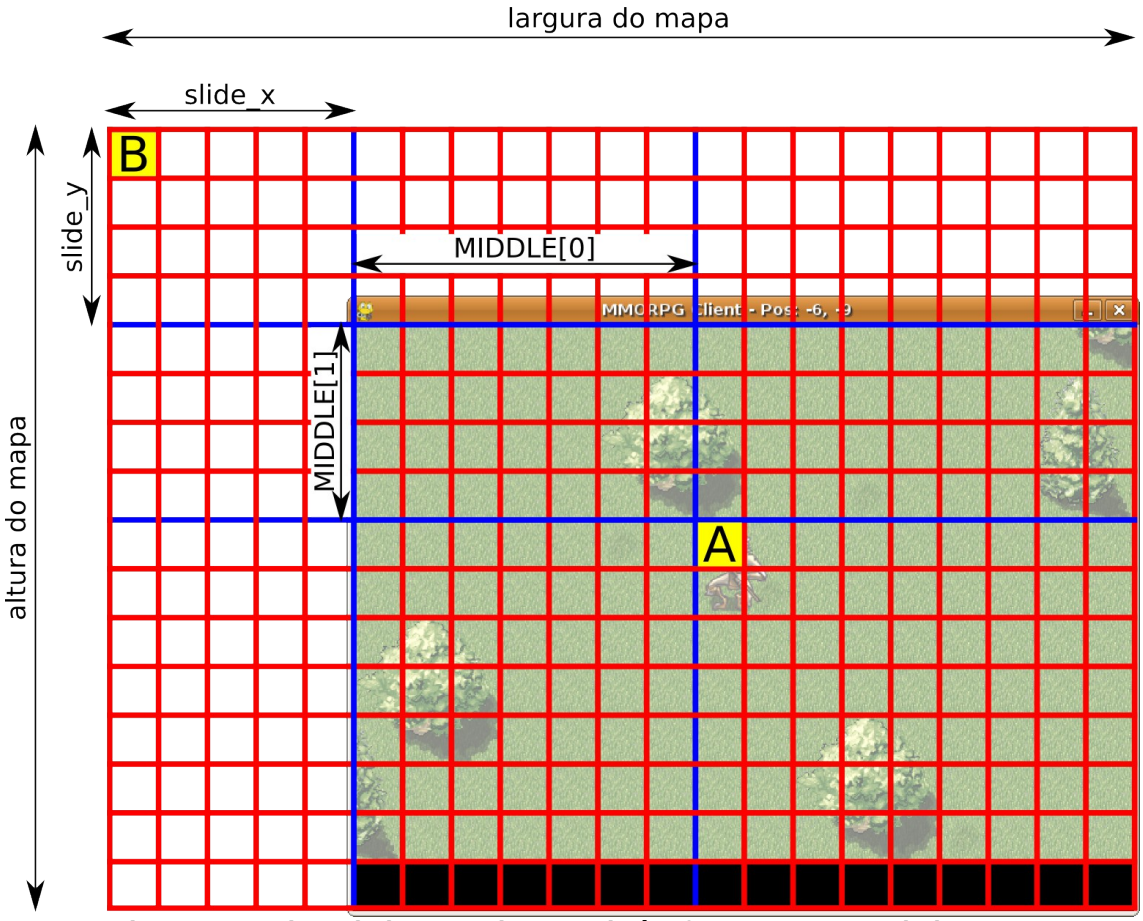

Figura 6: Matrizes do jogo. Toda a matriz é referente ao mapa do jogo, as variáveis slide_x e slide_y representam o deslocamento em relação ao ponto B (coordenada 0,0 do mapa), enquanto que a constante MIDDLE representa a coordenada de distância do canto superior esquerdo da tela até o personagem principal (ponto A).

1.5.4. Movimentando o personagem

Utilizando a função ***goto***, a movimentação do jogador é realizada por meio de uma chamada dessa função para a nova posição do personagem. Existem dois problemas que novos programadores de jogos normalmente enfrentam:

1. Movimentos coerentes, que consiste em fazer com que o personagem se movimente a uma certa velocidade e em todas as direções.
2. Aceleração e velocidade do personagem, pois o jogador espera que enquanto estiver pressionando a tecla direcional para a movimentação o personagem irá se movimentando a uma velocidade constante ou acelerando (mais comum em jogos de corrida).

Ambos os problemas podem ser solucionados com a abordagem a ser utilizada no jogo, que se baseia em uma variável com valores numéricos de velocidade do personagem, em que ele pode estar se movendo em uma determinada direção ou não. A velocidade é um valor entre -1, 0 e 1, em que cada valor determina a orientação do deslocamento do personagem em cada um dos dois eixos X e Y, como pode ser visto na Figura 7, seguindo a mesma orientação da Pygame para baixo e para a direita como eixos positivos.

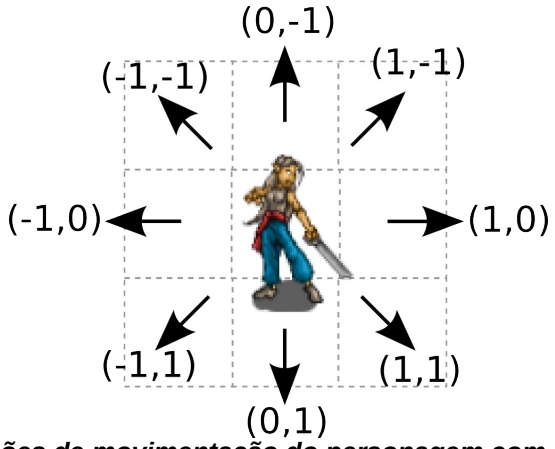

Figura 7: Direções de movimentação do personagem com as velocidades de deslocamento, considerando o eixo positivo para baixo e para a direita, no mesmo formato que a Pygame.

Então, a cada quadro executado no *loop*, o personagem irá se deslocar de acordo com a velocidade, sendo (0,0) a velocidade nula, em que o jogador não se movimenta. A Pygame possui constantes para mapear os botões do teclado, como as setas de direção. Como cada seta influencia diretamente na velocidade da direção, pode-se mapear para que cada seta seja um valor de velocidade de acordo com as direções da Figura 7:

```
MOVES = {
K_RIGHT: ( 1, 0),
K_LEFT : (-1, 0),
K_UP   : ( 0,-1),
K_DOWN : ( 0, 1)
}
```

A constante ***MOVES***[6] mapeia cada botão do teclado com a velocidade no eixo que o botão deve influenciar. Dessa maneira, enquanto o jogador estiver pressionando alguma tecla, o valor de velocidade dessa se somará à velocidade total do personagem, possibilitando movimentos diagonais ou que as velocidades em um eixo se anulem caso o jogador pressione setas de direções opostas.

Para concretizar o movimento, será criada outra função, com o intuito de processar o movimento e mandar o personagem para a posição. A função ***goto*** poderia já movimentar o personagem, porém o uso de outra função garante compatibilidade com o servidor já construído.

A função ***move*** processa apenas algum possível movimento do jogador e cria um evento do tipo ***GOTO*** para mandar o personagem para a nova posição, caso ele tenha movido:

```
from pygame.event import Event

def move((inc_x, inc_y)):
  if inc_x != 0 or inc_y != 0:
    e = Event(GOTO, {'x': slide_x + inc_x + MIDDLE[0], 'y': slide_y
+ inc_y + MIDDLE[1]})
```

6 Alguns programadores são contra o uso de variáveis e constantes no plural para simbolizar listas e dicionários, mas o plural pode evidenciar que se trata de uma lista ou um dicionário.

```
    pygame.event.post(e)
```

Se a velocidade não é (0,0), cria um novo evento ***GOTO***, que possui dois atributos: x e y. Cada atributo representa a coordenada de destino, e portanto é necessário somar a velocidade ao deslocamento da tela (***slide_x*** ou ***slide_y***) e o deslocamento do personagem em relação ao canto superior esquerdo da tela, em cada um dos eixos horizontal e vertical. Criado o evento, a função ***post*** adiciona o evento na fila da Pygame.

1.5.5. Manipulando eventos do jogo

Primeiramente, é necessário armazenar a velocidade atual do jogador, em uma variável ***moving***:

```
moving = (0,0)
```

Começando com valor nulo para o personagem permanecer parado. Então, cria-se uma função apenas para tratar eventos, ao invés de deixar o tratamento de eventos no *loop* e poluir o código. A função ***handle*** deverá cuidar de cada tipo de evento que vier, dos que forem interessante para serem processados:

```
def handle():
  global running, moving
  for e in pygame.event.get():
    if e.type == QUIT:
      running = False
    elif e.type == KEYDOWN:
      if e.key in MOVES.keys():
        moving = (moving[0] + MOVES[e.key][0], moving[1] +
MOVES[e.key][1])
    elif e.type == KEYUP:
      if e.key in MOVES.keys():
        moving = (moving[0] - MOVES[e.key][0], moving[1] -
MOVES[e.key][1])
    elif e.type == GOTO:
      goto(e.x, e.y)
```

O tratamento apresentado é referente a fechar o jogo, como já visto anteriormente, quando o jogador pressiona e solta uma tecla (***KEYDOWN*** e ***KEYUP*** respectivamente) e quando o sistema reconhece um evento do tipo ***GOTO***, que denomina onde o personagem deve se posicionar.

Quando pressionar alguma tecla, evento ***KEYDOWN***, uma estrutura ***if*** verifica se o botão pressionado (***e.key***) está entre as chaves do dicionário ***MOVES***. Caso esteja, o botão é uma das setas do teclado, então realiza o movimento de acordo com o valor atual de ***moving*** + o valor correspondente do dicionário ***MOVES***. Quando alguma tecla for solta, subtrai-se o referente valor de velocidade. No *loop* do jogo será necessário ainda processar a variável ***moving*** para concretizar o movimento.

O *loop*, com a movimentação do jogador, também fica diferente, pois agora a tela será renderizada em relação ao deslocamento da tela, além de ser necessário o tratamento de eventos:

```
while running:
```

```
  clock.tick(7)
  handle()
  move(moving)
  for y in range(HEIGHT):
    for x in range(WIDTH):
      i,j = (slide_x+x,slide_y+y)
      screen.blit(tile[tilemap[j][i]], (x*TILE, y*TILE))
  for y in range(-6,HEIGHT):
    for x in range(-4,WIDTH):
      i, j = (x+slide_x,slide_y+y)
      if objmap[j][i] < 24:
        screen.blit(obj[objmap[j][i]], (x*TILE, y*TILE))
  pygame.display.flip()
```

As variáveis ***i*** e ***j*** são na realidade a posição na tela que varia (***x*** e ***y*** do *loop*) somadas ao deslocamento da tela, a fim de colocar o objeto ou a textura da coordenada do mapa (i,j) na coordenada correta da tela (x,y). As chamadas na função ***handle*** e depois na ***move*** garantem a manipulação de eventos e movimentação do personagem.

1.5.6. Jogo com manipulação de eventos

Com a conclusão desta seção, tem-se um jogo capaz de apenas movimentar a tela, sem qualquer personagem, apesar da estrutura de centralizar o personagem já estar pronta. O código até o momento está apresentado no Quadro 1.

```
01| import pygame, cPickle
02| from pygame.locals import *
03| from pygame.event import Event
04| tilemap = cPickle.load(file('data/tilemap.txt', 'r'))
05| objmap = cPickle.load(file('data/objmap.txt', 'r'))
06| running = True
07| screen = pygame.display.set_mode((640,480))
08| clock = pygame.time.Clock()
09|
10| GOTO = 24
11| MESSAGE = 25
12| SETOBJECT = 26
13| SYSTEM = 27
14| GETINFO = 28
15|
16| WIDTH, HEIGHT, TILE = 16, 12, 40
17| MIDDLE = WIDTH/2 - 1 , HEIGHT/2 - 1
18| tile, obj = [], []
19|
20| for i in range(24):
21|   tile += [pygame.image.load(file('imagens/tile%s.png'%i))]
22|   obj += [pygame.image.load(file('imagens/obj%s.png'%i))]
23|
24| slide_x = 0
25| slide_y = 0
26|
27| def goto(x, y):
28|   global slide_x, slide_y
29|   slide_x = x - MIDDLE[0]
30|   slide_y = y - MIDDLE[1]
31|   pygame.display.set_caption("MMORPG Client - Pos: %2d, %2d"%(x,
y))
```

```
32|
33| def move((inc_x, inc_y)):
34|   if inc_x != 0 or inc_y != 0:
35|     e = Event(GOTO, {'x': slide_x + inc_x + MIDDLE[0], 'y':
slide_y + inc_y + MIDDLE[1]})
36|     pygame.event.post(e)
37|
38| MOVES = {
39| K_RIGHT: ( 1, 0),
40| K_LEFT : (-1, 0),
41| K_UP   : ( 0,-1),
42| K_DOWN : ( 0, 1)
43| }
44|
45| moving = (0,0)
46| def handle():
47|   global running, moving
48|   for e in pygame.event.get():
49|     if e.type == QUIT:
50|       running = False
51|     elif e.type == KEYDOWN:
52|       if e.key in MOVES.keys():
53|         moving = (moving[0] + MOVES[e.key][0], moving[1] +
MOVES[e.key][1])
54|     elif e.type == KEYUP:
55|       if e.key in MOVES.keys():
56|         moving = (moving[0] - MOVES[e.key][0], moving[1] -
MOVES[e.key][1])
57|     elif e.type == GOTO:
58|       goto(e.x, e.y)
59|
60| while running:
61|   clock.tick(7)
62|   handle()
63|   move(moving)
64|   for y in range(HEIGHT):
65|     for x in range(WIDTH):
66|       i,j = (slide_x+x,slide_y+y)
67|       screen.blit(tile[tilemap[j][i]], (x*TILE, y*TILE))
68|   for y in range(-6,HEIGHT):
69|     for x in range(-4,WIDTH):
70|       i, j = (x+slide_x,slide_y+y)
71|       if objmap[j][i] < 24:
72|         screen.blit(obj[objmap[j][i]], (x*TILE, y*TILE))
73|   pygame.display.flip()
```

Quadro 1: Código do jogo em versão intermediária.

1.6. Redes e conexão do cliente

A comunicação entre dispositivos computacionais tem se intensificado, através de redes de computadores que conectam os dispositivos entre si com o objetivo de compartilhar serviços. Para a comunicação, utilizam-se protocolos de rede, que definem o formato e a ordem das mensagens trocadas entre dois ou mais dispositivos computacionais bem como ações realizadas na transmissão e recebimento de uma mensagem (Kurose & Ross, 2005).

Um protocolo de rede muito utilizado, que também será utilizado pelo jogo, é o TCP (Kurose & Ross, 2005, p.178), que utiliza uma conexão orientada, ou seja, as aplicações que o utilizam trocam mensagens de controle para auxiliar no controle do fluxo de pacotes. Um pacote possui uma quantidade fixa ou variável de *bytes* para transitar de um dispositivo computacional a outro.

Dois modelos bastante comuns para utilização de rede são de cliente-servidor e *Peer-to-Peer* (P2P). Cliente-servidor baseia-se em sistemas conectando em um mesmo dispositivo, o servidor, que centraliza as informações. Redes P2P são sistemas de nós capazes de se interconectarem e se organizarem em topologias de redes (Androutsellis-Theotokis & Spinellis, 2004).

O modelo cliente-servidor será utilizado no jogo pois o servidor agirá como entidade centralizadora e controladora dos jogadores. A segurança em jogos online é um assunto que ainda recorre aos atuais jogos, pois falhas encontradas podem dar vantagens a um jogador, tornando o jogo desmotivante em um MMORPG, quando por exemplo um jogador consegue mais pontos apenas com uma falha.

Existem sempre pontos positivos e negativos em cada aspecto de segurança. Utilizar um servidor para validar movimentação e ações do jogador auxilia na coerência do jogo, já que em geral os servidores não podem ser alterados pelos jogadores, mas os clientes sim. O quanto se processa no servidor dependerá da capacidade que se espera utilizar do servidor em recursos computacionais e o quão comprometida a jogabilidade pode estar. Por exemplo, o jogador pode apenas falar para onde ele está indo e o servidor responder que ele pode, ou o jogador passar o valor do clique do mouse e o servidor responde onde ele vai.

Outro aspecto muito comum em mundos virtuais é utilização de *spoofing*, em que um jogador consegue se passar por outro, roubando itens virtuais, causando problemas sociais no mundo virtual (como por exemplo tomar decisões quando o jogador tem um cargo importante), dentre outros problemas. A utilização de um sistema de autenticação, nomes únicos para jogadores e até mesmo utilização de chaves assimétricas pode auxiliar no combate a esse tipo de prática.

Esta seção detalhará como criar conexões de *sockets* entre um cliente e um servidor, para conectar o cliente do jogo a um servidor.

1.6.1. Threads em jogos

Um assunto muito abordado principalmente na área de Sistemas Operacionais é a concorrência de processos. Os sistemas operacionais que são utilizados atualmente podem fazer diversas atividades ao mesmo tempo, e a utilização de processos possibilita isso. Cada processo utiliza recursos próprios, com espaço de endereçamento e de memória próprios (Tanenbaum & Woodhull, 2006, p.56).

O sistema operacional utiliza alguma maneira de alternar entre os processos para que todos possam ser executados em um curto período de tempo, tornando-os praticamente paralelos. Alguns sistemas operacionais atuais também utilizam mais que um processador para executar processos de fato em paralelo.

Dentro de um processo existe uma única *thread* por padrão, mas pode haver mais de uma a fim de compartilhar o mesmo espaço de endereçamento, ou outros recursos em comum. A grande diferença de processos e *threads* é que um processo é usado para agrupar recursos e *threads* são programadas para execução na CPU

(Tanenbaum & Woodhull, 2006, p.65), normalmente utilizando os mesmos recursos como a memória principal.

A utilização de *threads* em jogos é quase que fundamental, pois constantemente há o processamento de mais de um tipo de informação, normalmente sendo uma *thread* para renderização do vídeo/display e outra para manipular os eventos do usuário, como é o caso do cliente sendo construído. O servidor já se utiliza de diversas *threads*, uma para cada cliente a fim de manter a comunicação síncrona.

Para executar o serviço de receber mensagens do servidor e não travar o *loop* do jogo, é necessário criar uma *thread*. A manipulação de *threads* em Python pode ser feita com o módulo ***thread***:

```
>>> import thread
>>> dir(thread)
['LockType', '__doc__', '__name__', '_local', 'allocate',
'allocate_lock', 'error', 'exit', 'exit_thread', 'get_ident',
'interrupt_main', 'stack_size', 'start_new', 'start_new_thread']
```

A função ***start_new_thread*** inicializa uma *thread* usando uma função como parâmetro. No caso, a função fornecida será executada como uma *thread*. O segundo parâmetro é uma tupla com os parâmetros da função a ser executada em *thread*, que quando não há deve ser passada uma tupla vazia. A execução de uma função pode ser testada com o código:

```
>>> def teste():
...     print "hello mundo"
...
>>> thread.start_new_thread(teste, tuple())
-1211667568
hello mundo
```

Foi necessário utilizar a função ***tuple***, que retorna uma tupla vazia, para passar que não há parâmetros na função. O retornado antes da impressão do texto, o número que varia a cada chamada, é o código da *thread*, que pode ser utilizado para manipulá-la.

1.6.2. Conexão cliente-servidor com Sockets

Uma maneira interessante e fácil de criar um modelo de conexão cliente-servidor em Python é utilizando *socket*, que é uma porta entre o processo da aplicação e o protocolo de comunicação (Kurose & Ross, 2005, p.114). O *socket* abre a possibilidade de criar um canal de comunicação entre dispositivos, como por exemplo entre um cliente e um servidor.

Como exemplo de um bom uso de *sockets*, será construído um serviço de bate-papo para ilustrar o uso de servidor e cliente. O módulo Python ***socket*** cria um *socket* que pode ter como parâmetros o tipo de conexão, mas sem parâmetros cria uma conexão TCP padrão:

```
>>> import socket, thread
>>> server = socket.socket()
```

A diferença entre um servidor e um cliente é que um recebe a conexão e o outro tenta se conectar. Para receber a conexão, utiliza-se o método ***bind***, com um parâmetro de tupla de dois valores, sendo o primeiro o IP que recebe as conexões (pode ser uma *string* vazia) e o segundo a porta (recomenda-se utilizar uma porta acima de 1000, pois portas baixas são normalmente reservadas pelo sistema operacional):

```
>>> server = socket.socket()
>>> server.bind(("", 5050))
>>> server.listen(1)
```

O método ***listen*** inicializa o *socket* para que ele comece a esperar por conexões na porta fornecida. Agora, será definida uma função para rodar em uma *thread*, que receberá a conexão do cliente e imprimirá os textos recebidos na tela. Primeiro é necessário aceitar a conexão com o método ***accept***, que retornará um objeto com a conexão gerada pelo cliente e um objeto com informações (não será utilizado), e então é possível imprimir mensagens com o método ***recv***:

```
>>> def server_loop():
...     connection, info = server.accept()
...     msg = ""
...     while msg != "fechar":
...             msg = connection.recv(9999)
...             print "server " + msg
...
>>>
>>> thread.start_new_thread(server_loop, tuple())
-1211667568
```

Enquanto não for recebida uma mensagem “fechar”, o servidor recebe as mensagens do cliente. O parâmetro do método ***recv*** é o tamanho máximo de cada mensagem, o número máximo de *bytes* que serão lidos.

Agora será construído o cliente, que pode ser usado no mesmo terminal. Ao invés do método ***bind***, será utilizado agora o método ***connect***, que tem como parâmetro também uma tupla com o endereço e a porta[7]:

```
>>> client = socket.socket()
>>> client.connect(("localhost", 5050))
```

Basta agora utilizar o método ***send*** para enviar mensagens de texto. O método precisa como parâmetro apenas uma cadeia de caracteres, e retorna o número de *bytes* enviados:

```
>>> client = socket.socket()
>>> client.connect(("localhost", 5050))
>>> client.send("olá!")
5
server olá!
>>> client.send("testando denovo!")
server testando denovo!
16
```

7 agora coloca-se “localhost” como endereço, para conexão local. Poderia ser o IP de um servidor.

O uso de *threads* pode possibilitar ainda o servidor aceitar diversos clientes simultaneamente, bem como conexões P2P. Para o jogo em construção, será feita apenas uma conexão, mas que irá receber e enviar mensagens ao servidor.

1.6.3. Conectando o cliente MMORPG ao servidor

Esta subseção irá finalizar o cliente, e cada trecho de código aparece com a linha correspondente ao código presente no final desta seção. Primeiramente, é interessante definir uma constante com o endereço e porta para conexão por *socket* a ser realizada com o servidor:

```
 12| DESTINY = ("127.0.0.1", 5000)
```

O servidor espera que a primeira mensagem do cliente seja uma mensagem de apresentação, um evento da Pygame do tipo ***SYSTEM*** contendo o ***id*** com o código do personagem (deve variar entre 14 e 19, referente aos objetos que podem ser personagens) e o nome do jogador:

```
 15| MYID = Event(SYSTEM, {'id': 15, 'name': 'anonimo'})
```

Como o servidor não prevê alteração de nome e ID, pode ser uma constante.

A movimentação do personagem agora deverá ser validada pelo servidor. Isso ocorre para facilitar a criação do código e, principalmente, para que a validação dos movimentos dos jogadores seja controlada pelo servidor, que funciona como uma entidade supervisora para que não hajam movimentos fraudulentos, como por exemplo um jogador andar mais do que o possível ou atravessar paredes.

Alguns jogos utilizam as duas abordagens, a validação de movimentos tanto no servidor quanto no cliente, a fim de evitar o envio de mensagens desnecessárias ao servidor (por exemplo, se um personagem tenta andar para cima de uma árvore, esse movimento nem precisa ser enviado ao servidor, pois será negado).

O cliente apenas enviará o evento do tipo ***GOTO*** para o servidor, ao invés de processar localmente. O que será processado pela função ***goto*** local serão apenas as mensagens recebidas pelo servidor, que darão as coordenadas para onde o personagem de fato deve ir:

```
 45| def move((inc_x, inc_y)):
 46|   if inc_x != 0 or inc_y != 0:
 47|     e = Event(GOTO, {'x': slide_x + inc_x + MIDDLE[0], 'y':
slide_y + inc_y + MIDDLE[1]})
 48|     send(str(e))
```

A função ***send***, a ser construída, envia mensagens do tipo *string* para outro *socket*, pois *sockets* em Python trabalham com *strings*. Por isso, é preciso transformar o objeto do tipo *Event* da Pygame em uma cadeia de caracteres, com a função ***str***.

É necessário então criar a conexão com o servidor, mas antes disso deve-se criar um laço para receber mensagens do servidor, pois o recebimento de mensagens trava o processo do jogo e nem sempre é previsível quando o servidor vai mandar uma mensagem (como por exemplo se houvesse um recurso de bate-papo).

A função ***listener*** será criada para ser executada em uma *thread* e irá receber mensagens vindas do servidor. O servidor só envia dois tipos de mensagens: ou são mensagens de texto de algum problema do sistema (até mesmo erros de conexão) ou são eventos da Pygame, em uma lista de eventos. Por isso, a cada mensagem, deve ser processada tal lista em formato de cadeia de caracteres para uma lista Python (já que *sockets* de Python se comunicam por cadeias de caracteres), e colocado cada evento da lista na fila da Pygame:

```
50| def listener():
51|   while running:
52|     msg = conn.recv(999999)
53|     if msg != "":
54|       try:
55|         for m in eval(msg):
56|           pygame.event.post(m)
57|       except:
58|         print "Servidor: ",msg
```

O tamanho do *buffer* de entrada, 999999 bytes, garante que não venham mensagens quebradas em mais que uma parte, pois a mensagem retorna ao atingir os 999999 bytes (nunca atinge) ou quando acabar a mensagem. O método testa se ocorre erro ao tentar fazer com que a mensagem seja uma lista de eventos da Pygame, caso contrário imprime a mensagem do servidor no terminal.

É interessante também usar uma função para envio de mensagens, apenas para facilitar o código:

```
60| def send(event):
61|   conn.send(str(event))
```

A função ***send*** envia um evento da Pygame em uma *string* corretamente. O objeto ***conn*** pode ser inicializado logo abaixo, com a conexão ao servidor, o início do processo para recebimento de mensagens e envio da mensagem de apresentação do jogador:

```
63| conn = socket.socket()
64| conn.connect(DESTINY)
65| thread.start_new_thread(listener, tuple())
66| conn.send(str(MYID))
```

Dessa maneira o cliente já está se conectando corretamente ao servidor. Mas considerando o código já construído neste capítulo, são necessárias ainda algumas alterações no tratamento de eventos, já que as movimentações e novos eventos serão enviados pelo servidor.

1.6.4. Tratando eventos do servidor e concluindo o cliente

Uma função ainda(?) interessante de ser construída é a ***settext***, para imprimir um texto na faixa preta que foi deixada na parte debaixo do jogo. Para isso, será utilizado o módulo pygame.font (Pygame, 2008c) que cria um texto e renderiza em uma superfície Pygame (*Surface)*:

```
28| font = pygame.font.SysFont("default", 16)
29| def settext(msg):
```

```
30|   global text
31|   text = pygame.Surface((640, 40))
32|   text.blit(font.render(msg, True, (255,255,255)), (5,12))
33|
34|
35| settext("")
```

A linha 28 cria o objeto de fonte, utilizando a fonte padrão do sistema operacional com tamanho 16. A linha 31 cria a superfície tendo como parâmetro uma tupla com as dimensões em pixels. Na linha 32 o texto é então renderizado na superfície criada, com o método ***render*** que retorna o texto dado no primeiro parâmetro em uma outra superfície, sendo os parâmetros seguintes referentes a opção de se habilitar ou não *antialias* (contornos do texto melhorados) e uma tupla de três valores de cor no formato RGB (no caso é branco pois é o máximo de cada cor, que varia de 0 a 255). O posicionamento deve ser feito no ponto (5,12) da superfície ***text*** para que fique com boa centralização. Então utiliza-se uma chamada em branco ou com alguma mensagem de boas vindas ao jogador para inicializar a variável ***text***.

Encerrando então o cliente, resta o tratamento dos novos eventos que o servidor envia. O primeiro evento utilizará o clique do mouse, que o cliente enviará ao servidor o pedido, um evento do tipo ***GETINFO***, com as coordenadas do clique em relação ao mapa, para ter de retorno um evento do tipo ***MESSAGE*** com informação ao usuário:

```
88|     elif e.type == MOUSEBUTTONDOWN:
89|       pos = slide_x + e.pos[0]/TILE, slide_y + e.pos[1]/TILE
90|       send(Event(GETINFO, {'pos':pos}))
93|     elif e.type == MESSAGE:
94|       settext(e.msg)
```

Tratar os eventos do tipo ***MESSAGE*** com a função ***settext*** fará com que as mensagens sejam apresentadas na tela do jogador, na faixa preta da tela. O atributo ***pos*** possui as coordenadas em *pixels* do clique do mouse, que devem ser divididas pelo valor de ***TILE*** para se obter a posição na matriz de renderização, e somar-se ao deslocamento do mapa para ter a posição do objeto no mapa global. É preciso tratar ainda mensagens do tipo ***SYSTEM***, que se referem a mensagens ao sistema não interessantes ao jogador mas sim ao desenvolvedor, como informações de conexão ou tentativas de burlar o servidor:

```
95|     elif e.type == SYSTEM:
96|       print e.msg
```

Há também as mensagens do tipo ***SETOBJECT***. Essas mensagens vindas do servidor determinam que em alguma posição da matriz ***obj*** o valor de algum objeto deve mudar, seja para 255 (vazio) ou para algum personagem. Na realidade então, a movimentação do personagem vai ser sempre uma série de eventos do tipo ***SETOBJECT*** que determinam o que desaparece (quando um personagem andou) e o que aparece na tela do jogador:

```
97|     elif e.type == SETOBJECT:
98|       objmap[e.y][e.x] = e.obj
```

Resta apenas a renderização da superfície de texto no *loop* do jogo, com a posição referente aos 40 *pixels* finais da tela.:

```
113|    screen.blit(text, (0,440))
```

1.6.5. Cliente final do capítulo

Com todas as seções, a construção do jogo deve resultar em algo semelhante ao exibido na Figura 8, cujo código é exibido no Quadro 2.

Figura 8: Captura de tela do cliente MMORPG em versão final.

```
 1| import pygame, cPickle, thread, socket
 2| from pygame.locals import *
 3| pygame.init()
 4| from pygame.event import Event
 5|
 6| GOTO = 24 # levar o seu personagem para posicao do evento
 7| MESSAGE = 25 # mensagem para o cliente
 8| SETOBJECT = 26 # colocar um objeto no mapa
 9| SYSTEM = 27 # mensagem do sistema, provavelmente um erro
10| GETINFO = 28 # requisitar informacoes em um ponto do mapa
11|
12| DESTINY = ("127.0.0.1", 5000)
13| WIDTH, HEIGHT, TILE = 16, 11, 40
14| MIDDLE = WIDTH/2 - 1 , HEIGHT/2 - 1
15| MYID = Event(SYSTEM, {'id': 15, 'name': 'anonimo'})
```

```
 16|
 17| tilemap = cPickle.load(file('data/tilemap.txt', 'r'))
 18| objmap = cPickle.load(file('data/objmap.txt', 'r'))
 19| running = True
 20| screen = pygame.display.set_mode((640,480))
 21| clock = pygame.time.Clock()
 22| tile, obj = [], []
 23|
 24| for i in range(24):
 25|   tile += [pygame.image.load(file('imagens/tile%s.png'%i))]
 26|   obj += [pygame.image.load(file('imagens/obj%s.png'%i))]
 27|
 28| font = pygame.font.SysFont("default", 16)
 29| def settext(msg):
 30|   global text
 31|   text = pygame.Surface((640, 40))
 32|   text.blit(font.render(msg, True, (255,255,255)), (5,12))
 33|
 34|
 35| settext("")
 36| slide_x = 0
 37| slide_y = 0
 38|
 39| def goto(x, y):
 40|   global slide_x, slide_y
 41|   slide_x = x - MIDDLE[0]
 42|   slide_y = y - MIDDLE[1]
 43|   pygame.display.set_caption("MMORPG Client - Pos: %2d, %2d"%
(x, y))
 44|
 45| def move((inc_x, inc_y)):
 46|   if inc_x != 0 or inc_y != 0:
 47|     e = Event(GOTO, {'x': slide_x + inc_x + MIDDLE[0], 'y':
slide_y + inc_y + MIDDLE[1]})
 48|     send(str(e))
 49|
 50| def listener():
 51|   while running:
 52|     msg = conn.recv(999999)
 53|     if msg != "":
 54|       try:
 55|         for m in eval(msg):
 56|           pygame.event.post(m)
 57|       except:
 58|         print "Servidor: ",msg
 59|
 60| def send(event):
 61|   conn.send(str(event))
 62|
 63| conn = socket.socket()
 64| conn.connect(DESTINY)
 65| thread.start_new_thread(listener, tuple())
 66| conn.send(str(MYID))
 67|
 68| MOVES = {
 69|   K_RIGHT: ( 1, 0),
 70|   K_LEFT : (-1, 0),
 71|   K_UP   : ( 0,-1),
```

```
 72|   K_DOWN : ( 0, 1)
 73| }
 74|
 75| moving = (0,0)
 76|
 77| def handle():
 78|   global running, moving
 79|   for e in pygame.event.get():
 80|     if e.type == QUIT:
 81|       running = False
 82|     elif e.type == KEYDOWN:
 83|       if e.key in MOVES.keys():
 84|         moving = (moving[0] + MOVES[e.key][0], moving[1] +
MOVES[e.key][1])
 85|     elif e.type == KEYUP:
 86|       if e.key in MOVES.keys():
 87|         moving = (moving[0] - MOVES[e.key][0], moving[1] -
MOVES[e.key][1])
 88|     elif e.type == MOUSEBUTTONDOWN:
 89|       pos = slide_x + e.pos[0]/TILE, slide_y + e.pos[1]/TILE
 90|       send(Event(GETINFO, {'pos':pos}))
 91|     elif e.type == GOTO:
 92|       goto(e.x, e.y)
 93|     elif e.type == MESSAGE:
 94|       settext(e.msg)
 95|     elif e.type == SYSTEM:
 96|       print e.msg
 97|     elif e.type == SETOBJECT:
 98|       objmap[e.y][e.x] = e.obj
 99|
100| while running:
101|   clock.tick(7)
102|   handle()
103|   move(moving)
104|   for y in range(HEIGHT):
105|     for x in range(WIDTH):
106|       i,j = (slide_x+x,slide_y+y)
107|       screen.blit(tile[tilemap[j][i]], (x*TILE, y*TILE))
108|   for y in range(-6,HEIGHT):
109|     for x in range(-4,WIDTH):
110|       i, j = (x+slide_x,slide_y+y)
111|       if objmap[j][i] < 24:
112|         screen.blit(obj[objmap[j][i]], (x*TILE, y*TILE))
113|   screen.blit(text, (0,440))
114|   pygame.display.flip()
115|
116| conn.send("bye bye")
117| conn.close()
```

Quadro 2: Código fonte do cliente final MMORPG do capítulo.

1.7. Considerações finais

O objetivo deste capítulo foi construir um cliente MMORPG a fim de explorar alguns conceitos de desenvolvimento de jogos com a biblioteca Pygame e a plataforma Python. O projeto do jogo MMORPG pode ainda ser melhorado em diversos aspectos, bem

como derivado para outros gêneros de jogos que podem necessitar de algumas técnicas distintas.

Alguns módulos da Pygame possuem recursos muito interessantes, que podem ser explorados em profundidade com a documentação da Pygame oficial. Em especial, o módulo ***pygame.mixer*** manipula áudio para o jogo, ***pygame.sprite*** para manipular animações (para fazer por exemplo objetos animados), ***pygame.movie*** que permite a execução de vídeos, ***pygame.transform*** para trabalhar com edições em geral de *Surfaces*, além de módulos de entrada e saída como o ***pygame.joystick*** que permite uso de joysticks.

Uma constante dificuldade que novos programadores de Pygame encontram é a criação de interfaces gráficas, já que a Pygame por padrão não possui *widgets*, ou seja, elementos de interface gráfica prontos para uso, como caixa de diálogo para seleção de um arquivo, botões, menus dentre outras estruturas. Existem alguns *toolkits* direcionados a Pygame, sendo um de grande destaque a PGU (Phil Hassey, 2007), que utiliza Pygame nativo e suporta temas para a interface gráfica, contando com um grande número de *widgets*.

Referências

ANDROUTSELLIS-THEOTOKIS, S. and SPINELLIS, D. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 2004, vol. 36, p. 335–371.

ACHTERBOSCH, L., PIERCE, R., and SIMMONS, G.. Massively multiplayer online role-playing games: the past, present, and future. *Computers in Entertainment*, 2008, vol. 5, p. 1-33.

Blizzard. 2008. [On line]. *World of Warcraft*. Blizzard Enterteinement. Available from: <http://www.worldofwarcraft.com>. Access in: 2009 March 9.

CARPENTER, A., 2006. [On line]. Applying risk analysis to play-balance RPGs. In Gamasutra features. *Applying risk analysis to play-balance RPGs.* Available from:<http://www.gamasutra.com/features/20030611/carpenter_01.shtml>. Access in: 2009 March 6.

DOLL, S., 2002. [On line] *Get yourself into a Python cPickle*. Available from: <http://articles.techrepublic.com.com/5100-10878_11-1052190.html>. Access in: 2009 February 19.

Eletronic Arts Inc. 2008. [On line]. *Ultima Online*. Available from: < http://www.uoherald.com/news/>. Access in: 2009 March 7.

HASSEY, P., 2007. [On line]. *Phil's pyGame utilities*. Available from: <http://www.imitationpickles.org/pgu/wiki/index>. Access in: 2009 March 8.

HOLKNER, A., 2008. [On line] *Pyglet*. Available from: <http://www.pyglet.org/>. Access in: 2009 March 8.

KUROSE, JF. and ROSS, KW. *Redes de computadores e a Internet*: uma abordagem top-down. São Paulo: Person Addison Wesley, 2005. 656p.

Lucasfilm Ltda. 2002. [On line]. Take Part in a Thriving Star Wars Community. In *Star Wars Galaxies*. Available from: < http://starwarsgalaxies.station.sony.com/players/index.vm>. Access in: 2009 March 8.

LUTZ, M. and ASCHER, D. *Learning python*. Sebastopol, CA: O'Reilly \& Associates, Inc., 2003. 593p.

PYGAME. 2008a. [On line]. *Pygame documentation*: display. Available from: <http://www.pygame.org/docs/ref/display.html>. Access in: 2009 March 8.

PYGAME. 2008b. [On line]. *Pygame documentation*: event. [cited 8 March 2009]. Available from: <http://www.pygame.org/docs/ref/event.html>. Access in: 2009 March 8.

PYGAME. 2008c. [On line]. *Pygame documentation*: font. Available from: <http://www.pygame.org/docs/ref/font.html>. Access in: 2009 March 8.

PYGAME. 2008d. [On line]. *Pygame documentation*: image. Available from: <http://www.pygame.org/docs/ref/image.html>. Access in: 2009 March 8.

PYGAME. 2008e. [On line]. *Python game development*. Available from: <http://www.pygame.org/news.html>. Access in: 2008 November 29.

PYODE. 2007. [On line]. Available from: <http://pyode.sourceforge.net/>. Access in: 2009 March 8.

Python Foundation. 2009. [On line]. *Python programming language*. Available from: <http://www.python.org/>. Access in: 2009 February 19.

PYTHON-OGRE. 2009. [On line]. Available from: <http://www.python-ogre.org/>. Access in: 2009 March 8.

REIS, CR. 2004. [On line]. *Python na prática*: um curso objetivo de programação em Python. Available from: <http://www.async.com.br/projects/python/pnp/>. Access in: 2008 November 29.

ROCHA, HV. and BARANAUSKAS, MCC. *Design e avaliação de interfaces humano-computador*. Campinas: Universidade Estadual de Campinas - Unicamp, 2003.

Sony. 2008. [On line]. EverQuest – Massively Multiplayer Online Fantasy Role-Playing Game. Available from: <http://everquest.station.sony.com/>. Access in: 2009 March 8.

SOYA3D. 2008. [On line]. Available from: <http://home.gna.org/oomadness/en/soya3d/index.html>. Access in: 2009 March 8.

Square-Enix. 2008. [On line]. *Final Fantasy IX*. Available from: <http://na.square-enix.com/games/FFIX-gamesite/>. Access in: 2009 March 8.

TANENBAUM, A. and WOODHULL, A. *Operating Systems design and implementation*. [S.L.]: Pearson; Prentice Hall, 2006.

The Panda3D development team. 2008. [On line]. *Panda3D*. Available from: <http://panda3d.org/>. Access in: 2009 March 8.